EXHIBIT E

Packet Processor High Level Design

Revision 1.20

April 17, 2002

Octera

Principal Contributors:

Laury Flora
Paul Nunnally

Revision History

Revision	Date	Author	Description of Changes
0.10	July 31, 2001	Laury Flora	Initial document.
0.20	August 13, 2001	Laury Flora	Massive changes to the entire document.
0.30	August 21, 2001	Paul Nunnally	Added Control Word Format Added Implementation Section (to support Control Word Format) Added semantic section to Example Op-code Syntax Updated Interfaces Added Legend to Pseudo-code Syntax Reduced Event Registers from 9 to 6 Updated Read/Write definitions for Microcode Version Number Register Changed all "gate" references to "cell" Updated block diagram
1.00	August 28, 2001	Paul Nunnally	Added separate generic counters for each logical processor Added more details to status and configuration registers description Added error flag to the Initialization I/F Added additional description to the Implementation section Updated the op-code syntax (Appendix B) Corrected many other miscellaneous errors
1.01	August 30, 2001	Paul Nunnally	Added Verification Plan Removed "number of packet processors" issue, since this really is a question for the IPU HLD, which instantiates the packet processor Added missing compares to both MAC and VLAN hard compares Removed UDP checksum compare (no longer needed) Corrected subset for Flag Register 15
1.02	August 30, 2001	Paul Nunnally	Added IPU reference to the External Configuration Interface Added IPU reference to the Constant Mask and Data Registers description Corrected arithmetic constant selection description Corrected holding register data select microcode field name
1.03	September 11, 2001	Paul Nunnally	Updated Control Word Corrected descriptions in the Implementation Section Added SP Data field Added Abort Condition Register Modified Arithmetic Unit to be able to shift and add/sub on same cycle. Added Event Flag Set logic. Removed soft reset bit. Added Debug item to Open Issues. Added System Interface (clock and reset).
1.04	September 14, 2001	Paul Nunnally	Changed "drop_frame" to "sp_release". Added "option parse error" description to Option Parsing Unit. Added "option parse error" to hard compare result selection mux. Added statement about packet processor's robustness to packet header data. Added explaination of how Event Size field is to be set.
1.05	September 21, 2001	Paul Nunnally	Added Microcode Field Selection Tables. Updated Block Diagram.

Revision	Date	Author	Description of Changes
1.06	October 1, 2001	Paul Nunnally	Updated description for error signal on the Initialization Interface. Added Header Buffer Interface timing diagram. Added Event Queue Interface timing diagram. Added Initialization Interface timing diagram. Corrected control word bit assignments (Table 2-1). Corrected hex value in version register description. Corrected mac_frame_err_cmp description (Table 7-7). Corrected Astute Test description (Table 8-2). Added clarification to ipX_frm_end description.
1.07	October 3, 2001	Paul Nunnally	Updated Related Documents Table (Table 1-1). Removed debug register. Updated Register List (Table 6-1). Reduced event registers to 5 (register for MAC1 not needed). Redefined "ipX_frm_end" signal. Updated EQ timing diagram (Figure 7-4). Removed DEBUG register and added definition of "debug_out" bus.
1.08	October 17, 2001	Paul Nunnally	Renamed the microcode field LOAD_PAYLOAD_SCRATCH_OFFSET to LOAD_PS_OFFSET. Updated deseription for "ipX_wd_ena" in Table 4-3. Corrected SP Data Field description in section 8.13.7. Updated Payload Scratch Offset description in section 8.13.8. Updated Socket ID description in section 8.13.9. Removed Fabric Header Byte Fields, formerly section 8.13.17. Updated PE Condition 1 Selection Values in Table 8-39. Updated PE Condition 2 Selection Values in Table 8-40. Updated Figures 7-1, 7-3 and 7-4. Changed TMAC_UCAST_CMP and PMAC_UCAST_CMP hardcoded comparators to "not equal". Updated Flag Register 7 input subset in Table 8-17. Corrected Table 8-34 Corrected cross-reference for OP_OPTION_TYPE. Updated Event Registers (section 8.13). Updated EQ Write Control (section 8.20). Updated Related Documents Table (Table 1-1).
1.09	October 19, 2001	Paul Nunnally	Changed ETHER frame compare to 802.3 frame compare (see tables 8-7, 8-8 and 8-14). Updated Open Issues (section 10). Added to Summary (section 11). Added EQ_ADDR_SELECT to control word (Table 2-1). Added "Bytes Remaining" field to section 8.11 and Tables 8-20 & 8-24. Updated EQ Write Control (section 8.20). Added Bytes Remaining register description to section 8.21. Updated Estimated Cell Count (table 8-48).
1.10	October 30, 2001	Paul Nunnally	Corrected Flag Register 5 subset (Table 8-17). Added new input to CMP data selection mux (Table 8-4). Added new input to AU data selection mux (Table 8-20 and Table 8-24). Updated Option Parsing Unit description (Section 8.14). Updated Hardcoded SNAP compares (Table 8-9).

Revision	Date	Author	Description of Changes
1.11	November 5, 2001	Paul Nunnally	Reduce Flow Key Receive I/F field load to a single bit (Table 2-1 & Section 8.13.4) Added "divide by 16" capability to AU1 (Table 2-1 & Section 8.11). Corrected description in Section 3.3. Updated description for Register 6 (Table 3-8). Updated input subset for Flag Register 0 and 4(Table 8-18). Updated PE Condition 4 Selection Values in Table 8-42. Updated PE Condition 5 Selection Values in Table 8-43. Updated PE Condition 6 Selection Values in Table 8-44. Added TCP hard comparators (section 8.7.8). Corrected Create-Flow Figure. Updated description of Option Parsing Unit (Section 8.14).
1.12	November 12, 2001	Paul Nunnally	Updated ICMP Hard Compares (Table 8-12). Updated TCP Hard Compares (Table 8-13). Updated UDP Hard Compares (Table 8-14). Updated Hard Compare Mux Input Map (Table 8-15). Updated Option Parsing Unit description (Section 8.14). Updated Verification Plan (Section 9). Updated estimated cell count (Section 8.22).
1.13	November 20, 2001	Paul Nunnally	Extended Control Store to 428-bits (Sections 3.3.6 – 3.3.8, 8.18, 8.22). Updated Estimated Cell Count (Table 8-49). Removed "full_frame and "frame_err" ports (Sections 4.2, 8.21.4, 8.21.6). Corrected Initialization Interface Read Timing (Figure 7-2). Updated Control Word bit assignments (Table 2-1). Added "Length Abort Jump Address" Register (Section 3.3). Updated Abort Condition Unit (Section 8.10). Updated Priority Encoder Condition Data Selects (Tables 8-40 thru 8-45). Updated Control Store (Section 8.18). Updated Abort Condition Register (Section 8.21.2).
1.14	December 18, 2001	Paul Nunnally	Changed Checksum Accumulator Result from "valid" to "invalid" (Section 8.7.10 and 8.16). Updated Control Register (Section 3.3.5). Updated Version Register (Section 3.3.6). Updated Control Store RAM Address Register (Section 3.3.7). Updated Related Documents (Table 1-1). Updated Dual Logical Processor description (Section 2.3). Updated Feature List (Section 2.4). Corrected typo in Table 2-1. Updated Event Structure definition (Section 3.2). Updated Register Map (Section 3.3.1). Updated descriptions for Register 04 and Register 07 (Section 3.3). Added new ports for Flow Key Protocol masks (Sections 4.5 & 8.6). Changed MAC_SAP_CMP, VLAN_SAP_CMP, SNAP_UI_OUI_CMP and all ARP Hard Comparators to "not equal" (Sections 8.7.2 thru 8.7.5). Removed UDP_HDRLEN0_CMP (Sections 8.7.9 & 8,7.10). Added LHB halfword selection capability to EQ Data Selection Mux (Section 8.20). Updated Flow Key Field description (Section 8.13.4). Updated Option Byte Offset Fields description (Section 8.13.19). Updated Option Parsing Unit description (Section 8.14). Updated Verification Plan (Section 9). Changed all "input processor" references to "logical processor. Changed all "ipX_" references to "lpX_".

Revision	Date	Author	Description of Changes
1.15	December 20, 2001	Paul Nunnally	Updated Control Word (Section 2.7). Added additional descriptions for the Constant Mask Registers (Section 8.6). Updated Verification Plan (Section 9).
1.16	January 18, 2002	Paul Nunnally	Enhanced Register 01 description (Section 3.3.3). Added initialization sequence (Section 3.4). Corrected Event Flags description (Section 8.13.1). Updated Option Parsing Unit description (Section 8.14). Added "Corresponding Option Type Bit Map" column to Table 8-33. Updated description for Abort test (Table 9-2). Updated Related Documents Section (Table 1-1). Removed Socket ID Field description (formerly Section 8.13.9). Updated Event Queue description (Section 8.19). Updated HB Over-read Prevention Logic description (Section 8.21.5). Updated Header Side Info Register description (Section 8.21.6). Updated Test List (Table 9-2). Updated Option Parsing description (Section 8.14). Updated Checksum Accumulator description (Section 8.16). Updated Requirements description (Section 1.2). Updated Overview description (Section 1.3). Updated Definitions description (Section 1.4). Updated Basic Structure description (Section 2.2). Updated Feature List (Section 2.4). Updated Control Word (Table 2-1) Updated Tables in Section 3.3. Updated Header Buffer Interface descriptions (Table 4-2). Updated Event Queue Interface descriptions (Table 4-3). Updated Initialization Interface descriptions (Table 4-4). Updated External Configuration Interface descriptions (Table 4-5). Updated Design Rational descriptions (Section 5). Updated Design Parameters descriptions (Section 6). Removed Table from Section 8.5. Removed Table from Section 8.6. Updated General Purpose 16-bit Comparator description (Section 8.7.1). Updated Hardcoded Comparator descriptions (Sections 8.7.2 – 8.7.9). Updated Hardcoded Comparator Results Mux description (Section 8.7.10). Updated Event Flags description (Section 8.13.1). Updated Flow Key Field description (Section 8.13.4). Updated Control Store description (Section 8.18). Updated Estimated Cell Count (Table 8-47).
1.17	January 30, 2002	Paul Nunnally	Corrected ICMP_DATA_CMP compare equation (Table 8-10). Updated Frame ID Field description (Section 8.13.9). Updated IPU Number Field description (Section 8.13.13). Cleaned up Table 8-40. Updated Priority Branch test description (Table 9-2).
1.18	March 18, 2002	Paul Nunnally	Corrected bit assignment for the IPv4 More Fragments (MF) bit (Table 8-9). Updated the initialization sequence (Section 3.4). Added register defaults (Sections 3.3 & 3.3.5).
1.19		Paul Nunnally	Updated Header Buffer Interface Timing (Figure 7-3). Updated Hardcoded SNAP compares (Table 8-7). Updated Option Parsing Unit description (Section 8.14). Updated Checksum Accumulator description (Section 8.16).

Revision	Date	Author	Description of Changes
1.20	April 17, 2002	Paul Nunnally	Added new ports to Header Buffer Interface (Section 4.2). Updated SP Data Field description (Section 8.13.7). Updated Option Parsing Unit description (Section 8.14). Updated Header Side Info Register description (Section 8.21.6).

Table of Contents

List of Tables

List of Figures

1 Introduction

This is the High-Level-Design specification for the Astute VLIW Packet Processor.

As a VLIW processor, the packet processor can execute many functions in parallel, including the selection of data, mask and constant fields, making various comparisons to create condition flags, conditional branching, and data movement.

1.1 Related Documents

Document	Revision	Author
Content Processor Architectural Overview	0.41	Fazil Osman
Dispatcher High Level Design	1.25	Simon Knee
Flow Director CAM (FDC) High Level Design	1.27	Simon Knee
Input Processing Unit (IPU) High Level Design	1.20	Bob Sefton
LookUp Controller (LUC) High Level Design	1.23	Simon Knee
Message Bus High Level Design	1.22	Mark Feinstein
Protocol Cluster High Level Design	1.13	Kirk Larson
Queuing Model Trace of a Simple HTTP Request	1.00	Simon Knee
Some Statistical Plots of Web Traffic	1.00	Brian Petry
Scratchpad High Level Design	1.41	Charles Kaseff
TCP Processing Paths for the Content Processor	1.00	Simon Knee, Brian Petry
TCP/IP Algorithm Review	0.91	Brian Petry
TCP/IP Core Software: Functional Specification and Conformance Statement	1.10	Brian Petry
IPU to Processor Core Event Specification	1.16	Simon Knee

Table 1-1: Related Documents

1.2 Requirements

The entire packet processing system, consisting of a single Packet Processor, must be capable of processing a minimum-sized packet every 100-125ns. At 200MHz, that would be 20-25 clocks. Since the rest of the core is to run at 266MHz, the Packet Processor must also run at that rate. Furthermore, the area budget for the packet processing system is approximately 760,000 cells and 4 KB of instruction memory.

The Packet Processor must read and parse packets out of a Header Buffer, which contains the first 256 Bytes of a packet. It must write various pieces of information in an Event Queue. This includes data from the packet header, flags resulting from tests made on the packet header, checksums, etc.

1.3 Overview

The input to the packet processor consists of two structures: A Header Buffer which contains 256 up to Bytes, accessible as a FIFO in 16-byte quad-words, and a 43-bit Header Buffer Side Info register.

The output of the processor consists of an Event Queue, which can store up to 256 Bytes, written in 16-byte quad-words. This is accessed as an addressable RAM.

On a packet-by-packet basis, both the Header Buffer and Event Queue are FIFOs in the sense that the info for a packet can be popped (Header Buffer) or pushed (Event Queue), giving access to the next packet's info.

1.4 Definitions

Byte	8 bits
Half-Word	16 bits
Word	32 bits
Quad-Word	128 bits
VLIW	Very Long Instruction Word: A microprocessor architecture in which the instruction word controls many resources in parallel
Header Buffer	FIFO of sixteen 16-byte quad-words
Event Queue	Output storage of 16 16-byte quad-words
LHB	Little Header Buffer, a register holding up to 32-bytes of the Header Buffer, of which 16-bytes are always valid.
Microcode	The set of control words held in the instruction memory

2 Functional Operation

2.1 General Operation

Much of the functionality of the Packet Processor is defined by the Microcode. Like any processor, it could be programmed to do many different things. This section describes the hardware capabilities and features of the Packet Processor. All of these resources are under Microcode control.

The Packet Processor receives unmodified packet data (Host or Network) from the Header Buffer. The data is then processed as defined by the Microcode and produces, as a result, an Event Structure for the packet type that was processed. This Event Structure is then written into the Event Queue.

This flow is repeated for all packets.

2.2 Basic Structure

"Figure 2-1: Block Diagram" below is a diagram of the packet processor logic.

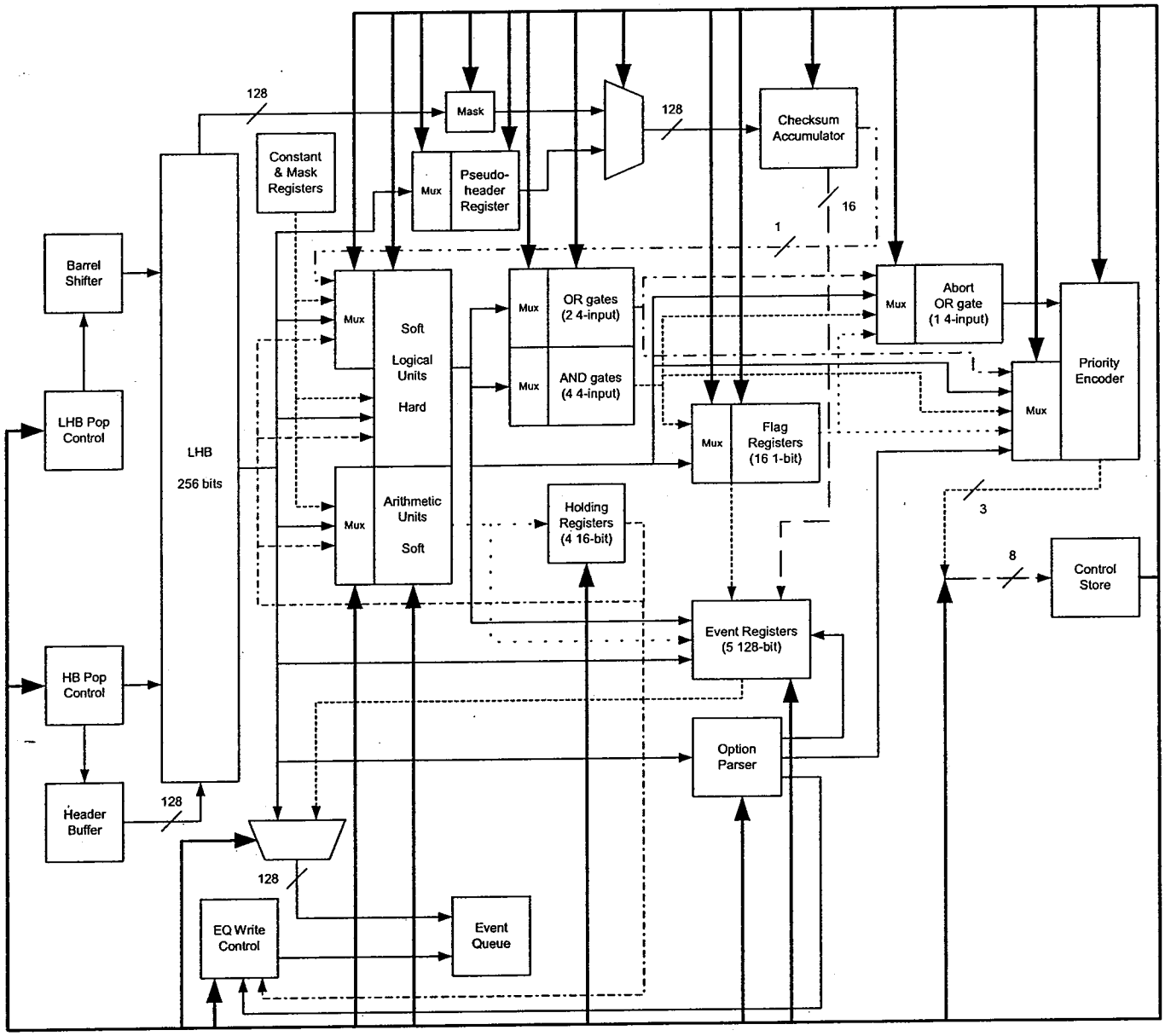

Figure 2-1: Block Diagram

For understanding purposes the Packet Processor can be broken into two main functional blocks: Control and Data.

Both the data and control paths are controlled by the Microcode (VLIW control word), which is an instruction read from the Control Store RAM. Also, both the data and control paths are fed by data from the Little

Header Buffer (LHB), which, in turn, is fed by the Header Buffer. The LHB acts as a FIFO. The "HB Pop Control" and "LHB Pop Control" shown in the block diagram together guarantee that at least the upper 16 bytes of data are always valid and available in the LHB. The Microcode can pop up to 16 bytes from the LHB, which will cause any remaining data to shift up to fill the newly-emptied positions. Simultaneously, if the pop will free up at least 16 bytes of empty space, then the "HB Pop Control" will write the next word from the Header Buffer into the LHB such that no gaps occur.

The control path selects data fields, masks those selected fields, and compares them to constants to produce various condition checks. These, in turn, can be ANDed or Ored to produce conditions which can be used immediately for conditional branching, loaded into flag registers for later use, or loaded into the Event Registers.

The data path also selects data fields, but instead of creating conditions, it can perform arithmetic operations, which in turn can be moved into various other places, such as a set of holding registers or the Event Registers. Additionally, data can be copied directly from the LHB to the Event Queue.

In addition, a checksum accumulator calculates the one's compliment checksum for all data popped from the LHB so that various protocol checksums can be verified.

2.3 Dual Logical Processor

In order to run the Packet Processor at the 266MHz core frequency, two clock cycles are required to execute an instruction. During the first clock cycle the instruction is fetched from the Control Store RAM. During the second clock cycle the instruction is executed, which includes calculating the address of the next instruction. For hardware optimization, a portion of the instruction execution will probably be moved into the first clock cycle, but that has no effect on understanding the design or writing code to be executed.

Since half of the design is unused during each clock cycle, the unused half can be used to execute an entirely separate instruction stream. While one instruction is being executed (the second clock of that instruction), another instruction from a separate instruction stream can be fetched from the Control Store RAM. This gives the effect of two logical processors.

Since this design will act as two independent packet processors, there needs to be two header buffers (with accompanying header side info), and two event queues. Also, all internal state needs to be duplicated. This includes the LHB, flag registers, event registers, and holding registers. Also, the Option Parsing Unit will be duplicated.

There is a single flip-flop called the Active Logical Processor which toggles between 0 and 1. This flip-flop controls which set of internal state is being accessed.

2.4 Feature List

The functionality of the Packet Processor may be understood by reading the pseudo-code (see Appendix A). This was written before the architecture described herein was generated, and was used as a guide to the architecture's requirements.

The VLIW Packet Processor can perform any or all of the following operations in a single instruction cycle:

- Indicate the completion of a packet, clearing various registers (such as the Pseudo-header), "pop"s the Header Buffer, and "push"s the Event Queue
- Perform many fixed compares of various fixed lengths and masks. These compares constitute most of what is necessary for packet processing. See Section 8.7 for a list of these hard-wired compares.
- Perform up to 4 general-purpose 16-bit compares, each of which may be "==", "<" or ">". The data for these comparators may be any contiguous group of 2 bytes in the upper 16 bytes of the LHB, or it may be one of the 4 Holding Registers. This data is masked by one of the Constant Mask Registers (selectable by Microcode) and then compared to one of the Constant Data Registers (selectable by Microcode).

- Any half-word (2 bytes) from the LHB, any Holding Register or the Immediate Data field, may be masked by a Constant Mask Register (similar to the compares) and then a Constant Data Register, or any Holding Register, may be added to or subtracted from this masked data. The result can be written into any holding register. Instead of addition or subtraction, the masked data can also be multiplied by 4 or divided by 16.
- Up to 4 compare results can be created by ANDing up to 4 outputs of the comparators.
- Up to 2 compare results can be created by ORing up to 4 outputs of the comparators.
- An Abort condition can be created by ORing up to 4 outputs of the comparators, flag registers, or outputs of the above AND/OR gates. See Section 8.10 for details.
- Any input to the above AND and OR gates may be inverted.
- A Create Flow flag can be created from the TCP flags.
- An Event Flag can be created by any of the 4 AND functions, any of the 2 OR functions, or directly from a comparator. See Section 8.13.1 for details.
- An Event Type and/or Event Subcode can be written with a constant from the control word.
- The Flow Key SP/DP, SA, DA and/or Protocol can be masked and loaded.
- The Flow Key Receive Interface can be loaded.
- The VLAN Tag field may be written.
- The Switch Fabric Header Length may be written.
- From 0 to 16 Bytes may be popped from the LHB, with the guarantee that there will be at least 16 valid Bytes in the LHB on the next clock.
- The Pseudo-header Register may be loaded with data from the LHB or a holding register.
- The checksum can be cleared.
- Data popped from the LHB or the Pseudo-header Register can be added to the checksum.
- The L3 header and Payload Scratch Offsets can be calculated and written.
- The actual packet length can be checked such that the packet processor does not overread the HB.
- The IP Option Length field may be written with the result of an arithmetic unit.
- The TCP Option Length field may be written with the result of an arithmetic unit.
- A hardware Option Parsing unit may be enabled, which will parse all of the options and then return control to the Microcode. Note that there are two copies of the Option Parsing unit, one for each logical processor.
- The Event Queue may be written with data from the LHB or any of the 5 Event Registers.
- The Microcode can jump to any other Microcode word.
- The Microcode can perform a prioritized conditional branch on up to 6 compare results plus the Abort condition.
- The Microcode can perform a 8-way "Case" branch on 3 compare results.

In order to reduce the silicon area, some restrictions may be placed on allowable combinations of operations, selection of LHB data position or constant register, etc.

2.5 Expected Performance

2.5.1 Packet Processor Performance Based on Minimum Ethernet Frame Metric

The packet processor must be able to sustain a stream of min-sized Ethernet frames at the wire speed of 5G. A min-sized Ethernet frame consists of 84 bytes (64 byte min frame + 20 byte overhead). This equates to 7,440,476 packets per second. Assuming a 266 MHz clock cycle, each packet must be processed in 35.75 clocks or less.
Pseudo code (see Appendix A) written for min-sized Ethernet packets (64 bytes) indicates that they can be processed in 14 instructions. Using a 266 MHz clock cycle and two logical processors interleaved on the same hardware, each logical processor will effectively operate at 133 MHz. Therefore, in 28 266 MHz clocks, two min-sized Ethernet frames can be processed, which is over 2.5x the required performance.

2.5.2 Packet Processor Performance Based on Connections Per Second Metric

The packet processor must be able to support 500,000 connections per second. A connection uses eight input packets (5 network + 3 host), yielding 4,000,000 packets per second, which means that a packet must be processed in 250 ns. At 266 MHz that is 66.5 clocks. Each dual-logical-processor packet processor will parse 2 min-sized Ethernet frames in 28 266 MHz clocks, or 14 clocks per packet, which is over 4.75x the required performance. Some of this headroom is desired for expansion and to allow for protocols that have not yet been examined.

2.5.3 Packet Processor Performance Based on Maximum Ethernet Frame Metric

The packet processor must be able to sustain a stream of IEEE max-sized Ethernet frames at the wire speed of 10G. A max-sized Ethernet frame consists of 1522 bytes. This equates to 821,287 packets per second. Assuming a 266 MHz clock cycle, each packet must be processed in 323.88 clocks or less. This is easily achieved if the two previous metrics are achieved.

2.6 Pseudo-Code

The functionality of the Packet Processor may be understood by reading the pseudo-code in Appendix A. This was written before the architecture described herein was generated, and was used as a guide to the architecture's requirements.

2.7 Control Word

The control word is a 428-bit instruction that specifies the exact operation of all the packet processor hardware for a given clock cycle. The exact implementation details for each of the control fields are described in Section 7. The individual fields of the control word are listed below in Table 2-1.

Field Name	Bit	Description
NEXT_ADDRESS	7:0	Read address for the control store.
EQ_WRITE_ADDRESS	11:8	Write address for the Event Queue.
EQ_WRITE_ENABLE	15:12	Word write enables for the Event Queue.
EQ_DATA_SELECT	19:16	Selects the data for a write to the Event Queue.
EQ_ADDR_SELECT	20	Selects the address for a write to the Event Queue.
LENGTH_ABORT_ENABLE	21	Enables aborts on a Byte Count Mismatch.
OP_OPTION_TYPE	22	Indicates which type of options is to be parsed by the Option Parsing Unit. Also determines destination event register for loading option offset and length.
OP_ENABLE	23	Enables the Option Parsing Unit.
LOAD_RUNNING_CSUM	24	Load enable for the Running Checksum field.

Field Name	Bit	Description
LOAD_IP_TCP_OPTION_OFFSET	25	Load enable for the IP or TCP Option Offset field.
LOAD_IP_TCP_OPTION_LENGTH	26	Load enable for the IP or TCP Option Length field.
IP_TCP_OPTION_LENGTH_DATA_SELECT	27	Selects the data to be loaded into the IP or TCP Option Length field.
INCREMENT_COUNTER	30:28	Increment enable for 1 of the General Purpose Counters.
LOAD_SF_HEADER_LENGTH	31	Load enable for the Switch Fabric Header Length field.
LOAD_VLAN_TAG	32	Load enable for the VLAN Tag (priority, CFI, VLAN ID) fields.
VLAN_TAG_DATA_SELECT	33	Selects the data to be loaded into the VLAN Tag fields.
LOAD_FLOW_KEY_PROTOCOL	34	Load enable for the flow key Layer 3 Protocol field.
LOAD_FLOW_KEY_SA_DA	35	Load enable for the flow key Layer 3 Source Address and Layer 3 Destination Address fields.
LOAD_FLOW_KEY_SPDP	36	Load enable for the flow key Layer 4 Source and Destination Ports field.
LOAD_PS_OFFSET	37	Load enable for the Payload Scratch Offset field.
LOAD_EVENT_SUBCODE	38	Load enable for the Event Subcode field.
LOAD_L3_HEADER_OFFSET	39	Load enable for the Layer 3 Header Offset field.
LOAD_IPU_SPI4_FRAME_ID	40	Load enable for the IPU Number, SPI4 Port Number and Frame ID fields.
LOAD_EVENT_SIZE	41	Load enable for the Event Size field.
LOAD_FLOW_KEY_RCV_IF	42	Load enable for the Receive Interface[field.
LOAD_EVENT_TYPE	43	Load enable for the Event Type field.
HOST_PACKET	44	Indicates which packet type is being processed.
LOAD_CREATE_FLOW_FLAG	45	Load enable for the Create Flow Flag field.
ER_EVENT_FLAG_ADDRESS	47:46	Destination Event Register Select.
LOAD_HOLDING_REG	51:48	Load enables for Holding Registers.
HOLDING_REG0_DATA_SELECT	52	Selects the data to be loaded into Holding Register 0.
HOLDING_REG1_DATA_SELECT	53	Selects the data to be loaded into Holding Register 1.
HOLDING_REG2_DATA_SELECT	54	Selects the data to be loaded into Holding Register 2.
HOLDING_REG3_DATA_SELECT	55	Selects the data to be loaded into Holding Register 3.
IMMEDIATE_DATA	71:56	Contains a value that can be used by the Arithmetic Units, Option Parsing Unit or Event Registers (size, type, subcode, option offset)
AU0_OPERATION_SELECT	73:72	Selects the operation to be performed by Arithmetic Unit 0.
AU0_DATA_SELECT	78:74	Selects the data to be used by Arithmetic Unit 0.
AU0_MASK_SELECT	81:79	Selects the mask to be used by Arithmetic Unit 0.
AU0_CONSTANT_SELECT	85:82	Selects the constant to be used by Arithmetic Unit 0.
AU1_OPERATION_SELECT	87:86	Selects the operation to be performed by Arithmetic Unit 1.
AU1_DATA_SELECT	92:88	Selects the data to be used by Arithmetic Unit 1.
AU1_MASK_SELECT	95:93	Selects the mask to be used by Arithmetic Unit 1.
AU1_CONSTANT_SELECT	99:96	Selects the constant to be used by Arithmetic Unit 1.
CA_RESET	100	Resets the Checksum Accumulator, clearing all registers.
CA_ENABLE	101	Enables the Checksum Accumulator, allowing it to calculate a new checksum.
CA_DATA_SELECT	102	Selects the data to be used by the Checksum Accumulator.
LOAD_PHDR_WORD0	106:103	Load enables for each byte of Word 0 of the Pseudo-header Register.
PHDR_WORD0_DATA_SELECT	109:107	Selects the data to be loaded into Word 0 of the Pseudo-header Register.
LOAD_PHDR_WORD1	113:110	Load enables for each byte of Word 1 of the Pseudo-header Register.
PHDR_WORD1_DATA_SELECT	116:114	Selects the data to be loaded into Word 1 of the Pseudo-header Register.
LOAD_PHDR_WORD2	120:117	Load enables for each byte of Word 2 of the Pseudo-header Register.

Field Name	Bit	Description
PHDR_WORD2_DATA_SELECT	123:121	Selects the data to be loaded into Word 2 of the Pseudo-header Register.
LOAD_PHDR_WORD3	127:124	Load enables for each byte of Word 3 of the Pseudo-header Register.
PHDR_WORD3_DATA_SELECT	130:128	Selects the data to be loaded into Word 3 of the Pseudo-header Register.
PE_BRANCH_TYPE	132:131	Indicates which type of branch is to be made by the Priority Encoder.
PE_CONDITION1_DATA_SELECT	138:133	Selects the data to be used for the Condition1 input of the Priority Encoder.
PE_CONDITION2_DATA_SELECT	144:139	Selects the data to be used for the Condition2 input of the Priority Encoder.
PE_CONDITION3_DATA_SELECT	150:145	Selects the data to be used for the Condition3 input of the Priority Encoder.
PE_CONDITION4_DATA_SELECT	156:151	Selects the data to be used for the Condition4 input of the Priority Encoder.
PE_CONDITION5_DATA_SELECT	162:157	Selects the data to be used for the Condition5 input of the Priority Encoder.
PE_CONDITION6_DATA_SELECT	168:163	Selects the data to be used for the Condition6 input of the Priority Encoder.
LOAD_FLAG_REG	184:169	Load enables for Flag Registers.
LOAD_EVENT_FLAG	200:185	Load enables for the Event Flag field.
SET_EVENT_FLAG	201	Enables the setting of the Event Flag field bits.
FLAG0_DATA_SELECT	203:202	Selects the data for Event Flag 0/Flag Register 0.
FLAG1_DATA_SELECT	205:204	Selects the data for Event Flag 1/Flag Register 1.
FLAG2_DATA_SELECT	207:206	Selects the data for Event Flag 2/Flag Register 2.
FLAG3_DATA_SELECT	209:208	Selects the data for Event Flag 3/Flag Register 3.
FLAG4_DATA_SELECT	211:210	Selects the data for Event Flag 4/Flag Register 4.
FLAG5_DATA_SELECT	213:212	Selects the data for Event Flag 5/Flag Register 5.
FLAG6_DATA_SELECT	215:214	Selects the data for Event Flag 6/Flag Register 6.
FLAG7_DATA_SELECT	217:216	Selects the data for Event Flag 7/Flag Register 7.
FLAG8_DATA_SELECT	219:218	Selects the data for Event Flag 8/Flag Register 8.
FLAG9_DATA_SELECT	221:220	Selects the data for Event Flag 9/Flag Register 9.
FLAG10_DATA_SELECT	223:222	Selects the data for Event Flag 10/Flag Register 10.
FLAG11_DATA_SELECT	225:224	Selects the data for Event Flag 11/Flag Register 11.
FLAG12_DATA_SELECT	227:226	Selects the data for Event Flag 12/Flag Register 12.
FLAG13_DATA_SELECT	229:228	Selects the data for Event Flag 13/Flag Register 13.
FLAG14_DATA_SELECT	231:230	Selects the data for Event Flag 14/Flag Register 14.
FLAG15_DATA_SELECT	233:232	Selects the data for Event Flag 15/Flag Register 15.
ABORT_INPUT0_DATA_SELECT	236:234	Selects the data for input 0 of the Abort OR Gate.
ABORT_INPUT1_DATA_SELECT	239:237	Selects the data for input 1 of the Abort OR Gate.
ABORT_INPUT2_DATA_SELECT	242:240	Selects the data for input 2 of the Abort OR Gate.
ABORT_INPUT3_DATA_SELECT	245:243	Selects the data for input 3 of the Abort OR Gate.
OR0_INPUT0_DATA_SELECT	249:246	Selects the data for input 0 of Condition OR Gate 0.
OR0_INPUT1_DATA_SELECT	253:250	Selects the data for input 1 of Condition OR Gate 0.
OR0_INPUT2_DATA_SELECT	257:254	Selects the data for input 2 of Condition OR Gate 0.
OR0_INPUT3_DATA_SELECT	261:258	Selects the data for input 3 of Condition OR Gate 0.
OR0_INVERSION_SELECT	265:262	Selects which inputs are to be inverted on Condition OR Gate 0.
OR1_INPUT0_DATA_SELECT	269:266	Selects the data for input 0 of Condition OR Gate 1.
OR1_INPUT1_DATA_SELECT	273:270	Selects the data for input 1 of Condition OR Gate 1.
OR1_INPUT2_DATA_SELECT	277:274	Selects the data for input 2 of Condition OR Gate 1.
OR1_INPUT3_DATA_SELECT	281:278	Selects the data for input 3 of Condition OR Gate 1.
OR1_INVERSION_SELECT	285:282	Selects which inputs are to be inverted on Condition OR Gate 1.
AND0_INPUT0_DATA_SELECT	289:286	Selects the data for input 0 of Condition AND Gate 0.
AND0_INPUT1_DATA_SELECT	293:290	Selects the data for input 1 of Condition AND Gate 0.
AND0_INPUT2_DATA_SELECT	297:294	Selects the data for input 2 of Condition AND Gate 0.
AND0_INPUT3_DATA_SELECT	301:298	Selects the data for input 3 of Condition AND Gate 0.

Field Name	Bit	Description
AND0_INVERSION_SELECT	305:302	Selects which inputs are to be inverted on Condition AND Gate 0.
AND1_INPUT0_DATA_SELECT	309:306	Selects the data for input 0 of Condition AND Gate 1.
AND1_INPUT1_DATA_SELECT	313:310	Selects the data for input 1 of Condition AND Gate 1.
AND1_INPUT2_DATA_SELECT	317:314	Selects the data for input 2 of Condition AND Gate 1.
AND1_INPUT3_DATA_SELECT	321:318	Selects the data for input 3 of Condition AND Gate 1.
AND1_INVERSION_SELECT	325:322	Selects which inputs are to be inverted on Condition AND Gate 1.
AND2_INPUT0_DATA_SELECT	329:326	Selects the data for input 0 of Condition AND Gate 2.
AND2_INPUT1_DATA_SELECT	333:330	Selects the data for input 1 of Condition AND Gate 2.
AND2_INPUT2_DATA_SELECT	337:334	Selects the data for input 2 of Condition AND Gate 2.
AND2_INPUT3_DATA_SELECT	341:338	Selects the data for input 3 of Condition AND Gate 2.
AND2_INVERSION_SELECT	345:342	Selects which inputs are to be inverted on Condition AND Gate 2.
AND3_INPUT0_DATA_SELECT	349:346	Selects the data for input 0 of Condition AND Gate 3.
AND3_INPUT1_DATA_SELECT	353:350	Selects the data for input 1 of Condition AND Gate 3.
AND3_INPUT2_DATA_SELECT	357:354	Selects the data for input 2 of Condition AND Gate 3.
AND3_INPUT3_DATA_SELECT	361:358	Selects the data for input 3 of Condition AND Gate 3.
AND3_INVERSION_SELECT	365:362	Selects which inputs are to be inverted on Condition AND Gate 3.
CMP0_OPERATION_SELECT	367:366	Selects the operation to be performed by General Purpose Comparator 0.
CMP0_DATA_SELECT	372:368	Selects the data to be used by General Purpose Comparator 0.
CMP0_MASK_SELECT	375:373	Selects the mask to be used by General Purpose Comparator 0.
CMP0_CONSTANT_SELECT	378:376	Selects the constant to be used by General Purpose Comparator 0.
CMP1_OPERATION_SELECT	380:379	Selects the operation to be performed by General Purpose Comparator 1.
CMP1_DATA_SELECT	385:381	Selects the data to be used by General Purpose Comparator 1.
CMP1_MASK_SELECT	388:386	Selects the mask to be used by General Purpose Comparator 1.
CMP1_CONSTANT_SELECT	391:389	Selects the constant to be used by General Purpose Comparator 1.
CMP2_OPERATION_SELECT	393:392	Selects the operation to be performed by General Purpose Comparator 2.
CMP2_DATA_SELECT	398:394	Selects the data to be used by General Purpose Comparator 2.
CMP2_MASK_SELECT	401:399	Selects the mask to be used by General Purpose Comparator 2.
CMP2_CONSTANT_SELECT	404:402	Selects the constant to be used by General Purpose Comparator 2.
CMP3_OPERATION_SELECT	406:405	Selects the operation to be performed by General Purpose Comparator 3.
CMP3_DATA_SELECT	411:407	Selects the data to be used by General Purpose Comparator 3.
CMP3_MASK_SELECT	414:412	Selects the mask to be used by General Purpose Comparator 3.
CMP3_CONSTANT_SELECT	417:415	Selects the constant to be used by General Purpose Comparator 3.
HARD_COMPARE_GROUP_SELECT	420:418	Selects the group of hard compare results to be used by the condition logic.
LHB_POP	425:421	Contains the number of bytes to be "popped" from the LHB.
SP_RELEASE	426	Indicates that the current frame being processed is to be released from Scratchpad Memory.

Field Name	Bit	Description
DONE	427	Indicates that the packet processor is finished processing the current frame. Firmware note: This bit MUST be asserted the next-to-last instruction of any processing sequence.

Table 2-1: Control Word Field Descriptions

3 Interfaces

3.1 Header Buffer

The Header Buffer contains a copy of the first 256 Bytes of a packet. The format of these packets is well described in the industry, and will not be repeated here.

3.2 Event Structure

An Event Structure is created for every packet that is parsed and is specific for each packet type. This is done by copying data directly from the LHB or extracting and modifying data from the LHB. The Packet Processor has 5 128-bit Event Registers which it uses to store information as it creates the various pieces of the Event Structure. The completed Event Structure is written into the Event Queue by the Packet Processor. The specific Event Structure for each packet type is defined in a separate document, as it is used throughout the chip.

The packet processor cannot drop an Event Structure under any circumstances. Instead, the Event Type field is set such that it will be discarded downstream.

3.3 Configuration and Status Registers

The following registers are accessible through the Initialization Interface.

Note that the Constant Data and Constant Mask Registers (see Section 4.5), which are accessible via Microcode control, are not shown here, since they reside outside the packet processor. Refer to the IPU HLD for details on these registers.

A base address (see IPU HLD) is required to access the Configuration and Status Registers of the packet processor. Any access attempted to an undefined register will result in an error being reported. For a read access the error is given coincident with the acknowledge. For a write the error is given the second clock cycle following the write attempt.

All register bits default to zero unless otherwise indicated.

3.3.1 Register Map

Address	Mode	Register Name	Description
00	Read Only	STATUS	Status Information
01	Read Clear	ERROR	Error Information
02	Read Write	ENABLE	Error Enables
03	Read Write	CONTROL	Control Information
04	Read Only	VERSION	Microcode Version Number
05	Read Write	RAM_ADDR	Control Store RAM Address
06	Read Write	RAM_DATA	Control Store RAM Data
07	Read Only	LENGTH_ABORT_ADDR	Length Abort Jump Address
08	Read Clear	COUNTER0_1	Generic Counter 1 for Logical Processor 0
09	Read Clear	COUNTER0_2	Generic Counter 2 for Logical Processor 0
0A	Read Clear	COUNTER0_3	Generic Counter 3 for Logical Processor 0
0B	Read Clear	COUNTER0_4	Generic Counter 4 for Logical Processor 0
0C	Read Clear	COUNTER0_5	Generic Counter 5 for Logical Processor 0
0D	Read Clear	COUNTER0_6	Generic Counter 6 for Logical Processor 0
0E	Read Clear	COUNTER0_7	Generic Counter 7 for Logical Processor 0
0F	Read Clear	COUNTER1_1	Generic Counter 1 for Logical Processor 1
10	Read Clear	COUNTER1_2	Generic Counter 2 for Logical Processor 1
11	Read Clear	COUNTER1_3	Generic Counter 3 for Logical Processor 1
12	Read Clear	COUNTER1_4	Generic Counter 4 for Logical Processor 1
13	Read Clear	COUNTER1_5	Generic Counter 5 for Logical Processor 1
14	Read Clear	COUNTER1_6	Generic Counter 6 for Logical Processor 1
15	Read Clear	COUNTER1_7	Generic Counter 7 for Logical Processor 1
16-1F	N/A	SPARE	N/A

Table 3-1: Packet Processor Register Map

3.3.2 Register 00: Status (STATUS)

Bits	Name	Description
0	STOPPED0	Indicates that Packet Processor 0 has stopped as a result of assertion of the STOP0 control bit
1	STOPPED1	Indicates that Packet Processor 1 has stopped as a result of assertion of the STOP1 control bit
2	OPT_PARSE0	Option Parsing Unit 0 is parsing options for logical processor 0
3	OPT_PARSE1	Option Parsing Unit 1 is parsing options for logical processor 1
4-31	N/A	N/A

Table 3-2: Status Register Bit Definitions

3.3.3 Register 01: Error (ERROR)

Bits	Name	Description
0	PAR_ERR	Control Store Parity Error bit. Cleared when read. This feature is not currently implemented, thus zero will always be read from this register.
1-31	N/A	N/A

Table 3-3: Error Register Bit Definitions

3.3.4 Register 02: Error Enable (ENABLE)

Bits	Name	Description
0	PAR_ERR_EN	Control Store Parity Error Enable bit. Enables the setting of the PAR_ERR bit in the ERROR register.
1-31	N/A	N/A

Table 3-4: Error Enable Register Bit Definitions

3.3.5 Register 03: Control (CONTROL)

Bits	Name	Description
0	STOP0	Requests that Packet Processor #0 stop at the end of the present packet. This bit defaults to '1'.
1	STOP1	Requests that Packet Processor #1 stop at the end of the present packet. This bit defaults to '1'.
2-3	RCVIFSRCSEL	Flow Key Receive Interface Source Select: 0 = all zeros 1 = VLAN ID[11:0] 2 = SPI-4 Port Number to bits 7:0, IPU ID to bit 8, zeros to bits 11:9. 3 = undefined
4-31	N/A	N/A

Table 3-5: Control Register Bit Definitions

3.3.6 Register 04: Version (VERSION)

Bits	Name	Description
0-23	VERSION	Microcode Version Number. This register is written by writing to the least significant 24-bits of the least significant word of Control Store location 255 (0xFF).
24-31	N/A	N/A

Table 3-6: Version Register Bit Definitions

3.3.7 Register 05: Control Store RAM Address

Bits	Name	Description
0-11	RAM_ADDR	Control Store RAM Address. The lower 4 bits indicate which 32-bit word is to be accessed, with "0000" meaning the least-significant word. The remaining 8 bits indicate which Control Store location is to be accessed.
12-31	N/A	N/A

Table 3-7: Control Store RAM Address Register

3.3.8 Register 06: Control Store RAM Data

Bits	Name	Description
0-31	RAM_DATA	Control Store RAM Data. Reading this register actually reads the Control Store RAM at the address indicated by RAM_ADDR[11:4]. Writing this register actually causes a write of the Control Store RAM, over-writing the 32-bit word indicated by RAM_ADDR[3:0]. NOTE: The Control Store RAM can be accessed only when both processors have been stopped.

Table 3-8: Control Store RAM Data Register

3.3.9 Register 07: Length Abort Jump Address

Bits	Name	Description
0-7	LENGTH_ABORT_ADDR	Length Abort Jump Address. This register contains the jump address due to a Byte Count Mismatch. It must point to the first instruction to be executed immediately following a Byte Count Mismatch. This register is written by writing to the most significant 8-bits of the least significant word of Control Store location 255 (0xFF).
8-31	N/A	N/A

Table 3-9: Length Abort Jump Address Register

3.3.10 Register 08: Generic Counter 1 for Logical Processor 0

Bits	Name	Description
0-31	GEN_CNTR0_1	Generic Counter 1 for Logical Processor 0. Reading this register causes it to clear. When LP0 is active and the Microcode field INCREMENT_COUNTER equals 001, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-10: Generic Counter 1 for Logical Processor 0 Register

3.3.11 Register 09: Generic Counter 2 for Logical Processor 0

Bits	Name	Description
0-31	GEN_CNTR0_2	Generic Counter 2 for Logical Processor 0. Reading this register causes it to clear. When LP0 is active and the Microcode field INCREMENT_COUNTER equals 010, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-11: Generic Counter 2 for Logical Processor 0 Register

3.3.12 Register 0A: Generic Counter 3 for Logical Processor 0

Bits	Name	Description
0-31	GEN_CNTR0_3	Generic Counter 3 for Logical Processor 0. Reading this register causes it to clear. When LP0 is active and the Microcode field INCREMENT_COUNTER equals 011, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-12: Generic Counter 3 for Logical Processor 0 Register

3.3.13 Register 0B: Generic Counter 4 for Logical Processor 0

Bits	Name	Description
0-31	GEN_CNTR0_4	Generic Counter 4 for Logical Processor 0. Reading this register causes it to clear. When LP0 is active and the Microcode field INCREMENT_COUNTER equals 100, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-13: Generic Counter 4 for Logical Processor 0 Register

3.3.14 Register 0C: Generic Counter 5 for Logical Processor 0

Bits	Name	Description
0-31	GEN_CNTR0_5	Generic Counter 5 for Logical Processor 0. Reading this register causes it to clear. When LP0 is active and the Microcode field INCREMENT_COUNTER equals 101, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-14: Generic Counter 5 for Logical Processor 0 Register

3.3.15 Register 0D: Generic Counter 6 for Logical Processor 0

Bits	Name	Description
0-31	GEN_CNTR0_6	Generic Counter 6 for Logical Processor 0. Reading this register causes it to clear. When LP0 is active and the Microcode field INCREMENT_COUNTER equals 110, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-15: Generic Counter 6 for Logical Processor 0 Register

3.3.16 Register 0E: Generic Counter 7 for Logical Processor 0

Bits	Name	Description
0-31	GEN_CNTR0_7	Generic Counter 7 for Logical Processor 0. Reading this register causes it to clear. When LP0 is active and the Microcode field INCREMENT_COUNTER equals 111, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-16: Generic Counter 7 for Logical Processor 0 Register

3.3.17 Register 0F: Generic Counter 1 for Logical Processor 1

Bits	Name	Description
0-31	GEN_CNTR1_1	Generic Counter 1 for Logical Processor 1. Reading this register causes it to clear. When LP1 is active and the Microcode field INCREMENT_COUNTER equals 001, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-17: Generic Counter 1 for Logical Processor 1 Register

3.3.18 Register 10: Generic Counter 2 for Logical Processor 1

Bits	Name	Description
0-31	GEN_CNTR1_2	Generic Counter 2 for Logical Processor 1. Reading this register causes it to clear. When LP1 is active and the Microcode field INCREMENT_COUNTER equals 010, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-18: Generic Counter 2 for Logical Processor 1 Register

3.3.19 Register 11: Generic Counter 3 for Logical Processor 1

Bits	Name	Description
0-31	GEN_CNTR1_3	Generic Counter 3 for Logical Processor 1. Reading this register causes it to clear. When LP1 is active and the Microcode field INCREMENT_COUNTER equals 011, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-19: Generic Counter 3 for Logical Processor 1 Register

3.3.20 Register 12: Generic Counter 4 for Logical Processor 1

Bits	Name	Description
0-31	GEN_CNTR1_4	Generic Counter 4 for Logical Processor 1. Reading this register causes it to clear. When LP1 is active and the Microcode field INCREMENT_COUNTER equals 100, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-20: Generic Counter 4 for Logical Processor 1 Register

3.3.21 Register 13: Generic Counter 5 for Logical Processor 1

Bits	Name	Description
0-31	GEN_CNTR1_5	Generic Counter 5 for Logical Processor 1. Reading this register causes it to clear. When LP1 is active and the Microcode field INCREMENT_COUNTER equals 101, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-21: Generic Counter 5 for Logical Processor 1 Register

3.3.22 Register 14: Generic Counter 6 for Logical Processor 1

Bits	Name	Description
0-31	GEN_CNTR1_6	Generic Counter 6 for Logical Processor 1. Reading this register causes it to clear. When LP1 is active and the Microcode field INCREMENT_COUNTER equals 110, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-22: Generic Counter 6 for Logical Processor 1 Register

3.3.23 Register 15: Generic Counter 7 for Logical Processor 1

Bits	Name	Description
0-31	GEN_CNTR1_7	Generic Counter 7 for Logical Processor 1. Reading this register causes it to clear. When LP1 is active and the Microcode field INCREMENT_COUNTER equals 111, this register will increment by 1. In the event that the maximum value is reached, it will NOT rollover.

Table 3-23: Generic Counter 7 for Logical Processor 1 Register

3.4 Initialization Sequence

The initialization sequence for the packet processor is as follows:

- Write the Control Register (set STOP bits)
- Load the External Configuration Registers (see IPU HLD for details)
- Load the Control Store
- Write the Control Register (clear STOP bits and set RCVIFSRCSEL field)

4 External Interface Signals

4.1 System Interface

Signal Name	Direction	Description
clock	Input	System Clock. Max Frequency is 266 MHz.
reset	Input	System Reset. Synchronous and active high.

Table 4-1: System Interface Pins

4.2 Header Buffer Interface

Signal Name	Direction	Description
lp0_hdr_data[127:0]	Input	Logical Processor 0 Header FIFO read port.
lp0_hdr_available	Input	Logical Processor 0 Packet Header Available. High when a complete header is available to be processed, the first word of the header is valid on hdr_data, and all side information is valid. Goes low after first FIFO read.
lp0_pvid[11:0]	Input	Logical Processor 0 Port VLAN ID of the current frame.
lp0_frame_id[7:0]	Input	Logical Processor 0 Frame ID of current frame.
lp0_port_num[7:0]	Input	Logical Processor 0 Destination SPI-4 port number for current frame.
lp0_event_num[3:0]	Input	Logical Processor 0 Event Sequence Number for current frame.
lp0_code_entry_point[2:0]	Input	Logical Processor 0 Configuration item passed from IRM. Selects port-specific code offset for PP.
lp0_byte_cnt[7:0]	Input	Logical Processor 0 received byte count of current frame. A byte count of 256 is indicated by 0x00.
lp0_force_sp_data	Input	Logical Processor 0 force SP Data indicator. When '1', the PP will force the SP Data bit in Event Register 1 to the value indicated by lp0_sp_data_val.
lp0_sp_data_val	Input	Logical Processor 0 SP Data force value.
lp0_hdr_fifo_rd	Output	Logical Processor 0 Read Strobe. Advances header FIFO read pointer. Current word should be latched before pointer is advanced (i.e. the first word in the FIFO is valid).
lp0_pp_done	Output	Asserted when Logical Processor 0 has finished processing the current header. Driven by microcode field DONE.
lp0_sp_release	Output	Logical Processor 0 Scratch Pad Release. Valid only when pp_done is high. If high, a release command will be issued to the Scratchpad for the frame just processed. Driven by microcode field SP_RELEASE.
lp1_hdr_data[127:0]	Input	Logical Processor 1 Header FIFO read port.
lp1_hdr_available	Input	Logical Processor 1 Packet Header Available. High when a complete header is available to be processed, the first word of the header is valid on hdr_data, and all side information is valid. Goes low after first FIFO read.
lp1_pvid[11:0]	Input	Logical Processor 1 Port VLAN ID of the current frame.
lp1_frame_id[7:0]	Input	Logical Processor 1 Frame ID of current frame.
lp1_port_num[7:0]	Input	Logical Processor 1 Destination SPI-4 port number for current frame.
lp1_event_num[3:0]	Input	Logical Processor 1 Event Sequence Number for current frame.
lp1_code_entry_point[2:0]	Input	Logical Processor 1 Configuration item passed from IRM. Selects port-specific code offset for PP.

lp1_byte_cnt[7:0]	Input	Logical Processor 1 received byte count of current frame. A byte count of 256 is indicated by 0x00.
lp1_force_sp_data	Input	Logical Processor 1 force SP Data indicator. When '1', the PP will force the SP Data bit in Event Register 1 to the value indicated by lp1_sp_data_val.
lp1_sp_data_val	Input	Logical Processor 1 SP Data force value.
lp1_hdr_fifo_rd	Output	Logical Processor 1 Read Strobe. Advances header FIFO read pointer. Current word should be latched before pointer is advanced (i.e. the first word in the FIFO is valid).
lp1_pp_done	Output	Asserted when Logical Processor 1 has finished processing the current header. Driven by microcode field DONE.
lp1_sp_release	Output	Logical Processor 1 Scratch Pad Release. Valid only when pp_done is high. If high, a release command will be issued to the Scratchpad for the frame just processed. Driven by microcode field SP_RELEASE.

Table 4-2: Header Buffer Interface Pins

4.3 Event Queue Interface

Signal Name	Direction	Description
eq0_ready	Input	Event Queue 0 Ready. High indicates a frame buffer is available.
lp0_waddr[7:0]	Output	Logical Processor 0 write address bus.
lp0_wvalid	Output	Logical Processor 0 write valid signal. High indicates valid write cycle.
lp0_wd_ena[3:0]	Output	Logical Processor 0 word enables. Valid when *lp0_wvalid* is high. Bit 0 enables lp0_wdata[31:0]. Bit 1 enables lp0_wdata[63:32]. Bit 2 enables lp0_wdata[95:64]. Bit 3 enables lp0_wdata[127:96].
lp0_wdata[127:0]	Output	Logical Processor 0 write data bus. Valid when *lp0_wvalid* is high.
lp0_frm_end	Output	Logical Processor 0 frame end signal. High indicates the input processor is finished with the current frame buffer.
eq1_ready	Input	Event Queue 1 Ready. High indicates a frame buffer is available.
lp1_waddr[7:0]	Output	Logical Processor 1 write address bus.
lp1_wvalid	Output	Logical Processor 1 write valid signal. High indicates valid write cycle.
lp1_wd_ena[3:0]	Output	Logical Processor 1 word enables. Valid when *lp1_wvalid* is high. Bit 0 enables lp1_wdata[31:0]. Bit 1 enables lp1_wdata[63:32]. Bit 2 enables lp1_wdata[95:64]. Bit 3 enables lp1_wdata[127:96].
lp1_wdata[127:0]	Output	Logical Processor 1 write data bus. Valid when *lp1_wvalid* is high.
lp1_frm_end	Output	Logical Processor 1 frame end signal. High indicates the input processor is finished with the current frame buffer.

Table 4-3: Event Queue Interface Pins

4.4 Initialization Interface

Signal Name	Direction	Description
cs	Input	Chip select signal generated by input processor unit
addr[4:0]	Input	Address to access 32-bit word data
r_wL	Input	Read/write bar. Write is active low and Read is active high
wr_data[31:0]	Input	Write data
ack	Output	Acknowledgement after putting valid data on rd_data during a read transaction
error	Output	Error indicator due to an illegal register access attempt.
rd_data[31:0]	Output	Read data

debug_out[6:0]	Output	Selected debug output from packet processor. The bit assignments are: Bit 6 – active_lp Bit 5 – lp1_op_parsing Bit 4 -- lhb1_ready Bit 3 – lp1_stopped Bit 2 – lp0_op_parsing Bit 1 -- lhb0_ready Bit 0 – lp0_stopped

Table 4-4: Initialization Interface Pins

4.5 External Configuration Interface

These inputs, with the exception of IPU_ID, are the Constant Mask and Data values used by the packet processor. The actual registers reside in the IPU (see IPU HLD for details), not the packet processor.

Signal Name	Direction	Description
ipu_id	Input	IPU ID
exp_pmac_da[47:0]	Input	Expected Pass-through MAC Destination Address
pmac_mask[47:0]	Input	Mask to be applied to DA before comparing to exp_pmac_da
exp_tmac_da[47:0]	Input	Expected Termination MAC Destination Address
tmac_mask[47:0]	Input	Mask to be applied to DA before comparing to exp_tmac_da
ip_option_types[23:0]	Input	3 possible IP option types (for option parsing unit)
tcp_option_types[79:0]	Input	10 possible TCP option types (for option parsing unit)
exp_protocol[15:0]	Input	Expected protocol value
exp_l3a_len[7:0]	Input	Expected L3 address length
exp_tcp_flags[5:0]	Input	Expected TCP flags
cf_mask[5:0]	Input	Mask to be applied to TCP Flags before comparing to exp_tcp_flgas
lp0_flow_key_spdp_mask[31:0]	Input	Selected mask for LHB SP/DP field before writing into Event Structure
lp0_flow_key_sa_mask[31:0]	Input	Selected mask for LHB SA field before writing into Event Structure
lp0_flow_key_da_mask[31:0]	Input	Selected mask for LHB DA field before writing into Event Structure
lp0_flow_key_ptcl_mask[7:0]	Input	Selected mask for LHB Protocol field before writing into Event Structure for Logical Processor 0
lp0_flow_key_ptcl_force[7:0]	Input	Selected force for LHB Protocol field before writing into Event Structure for Logical Processor 0
lp1_flow_key_spdp_mask[31:0]	Input	Selected mask for LHB SP/DP field before writing into Event structure for Logical Processor 1
lp1_flow_key_sa_mask[31:0]	Input	Selected mask for LHB SA field before writing into Event structure for Logical Processor 1
lp1_flow_key_da_mask[31:0]	Input	Selected mask for LHB DA field before writing into Event structure for Logical Processor 1
lp1_flow_key_ptcl_mask[7:0]	Input	Selected mask for LHB Protocol field before writing into Event Structure for Logical Processor 1
lp1_flow_key_ptcl_force[7:0]	Input	Selected force for LHB Protocol field before writing into Event Structure for Logical Processor 1
constant_data0[15:0]	Input	Constant data value
constant_data1[15:0]	Input	Constant data value
constant_data2[15:0]	Input	Constant data value
constant_data3[15:0]	Input	Constant data value
constant_data4[15:0]	Input	Constant data value
constant_data5[15:0]	Input	Constant data value
constant_data6[15:0]	Input	Constant data value

constant_data7[15:0]	Input	Constant data value
constant_data8[15:0]	Input	Constant data value
constant_data9[15:0]	Input	Constant data value
constant_data10[15:0]	Input	Constant data value
constant_data11[15:0]	Input	Constant data value
constant_data12[15:0]	Input	Constant data value
constant_data13[15:0]	Input	Constant data value
constant_data14[15:0]	Input	Constant data value
constant_mask0[15:0]	Input	Constant mask value
constant_mask1[15:0]	Input	Constant mask value
constant_mask2[15:0]	Input	Constant mask value
constant_mask3[15:0]	Input	Constant mask value
constant_mask4[15:0]	Input	Constant mask value
constant_mask5[15:0]	Input	Constant mask value
constant_mask6[15:0]	Input	Constant mask value

Table 4-5: External Configuration Interface Pins

5 Design Rational

5.1 General Principles

The main design trade-off is the number of operations to be performed in a single clock cycle. One extreme is a RISC design which can perform only one operation per clock, such as load, store, condition creation, or conditional branch.. The other extreme is to design special-purpose hardware which can do everything necessary in the fewest clock cycles possible (theoretically, it COULD be all done in a single clock cycle, although that might be a very slow clock cycle).

At the first extreme the design is very soft, and can probably be programmed to perform nearly any task. Unfortunately, it requires many clock cycles to perform the task.

At the other extreme the performance can theoretically be made as high as necessary. Unfortunately, the closer one approaches this extreme, the more cells required and the less soft the design is, making it less capable of being modified via programming.

5.2 Architecture Design Process

The Packet Processor started as a fairly standard RISC design, which suffered from the performance issue described above. To solve this performance issue, a very soft VLIW processor was proposed, which contained an abundance of comparators to feed the conditional branch logic. Each of these comparators had complete flexibility as to which LHB bytes and which constant registers were to be used, both to mask the selected LHB bytes and as data to be compared to the masked LHB bytes.

Synopsys was used to optimize some test code written to approximate the proposed design. The constraints were set to obtain the fastest achievable cycle time. The result suffered from a cell-count issue. To understand this issue, the design was optimized to various cycle times, producing the graph in Figure 5-1 below.

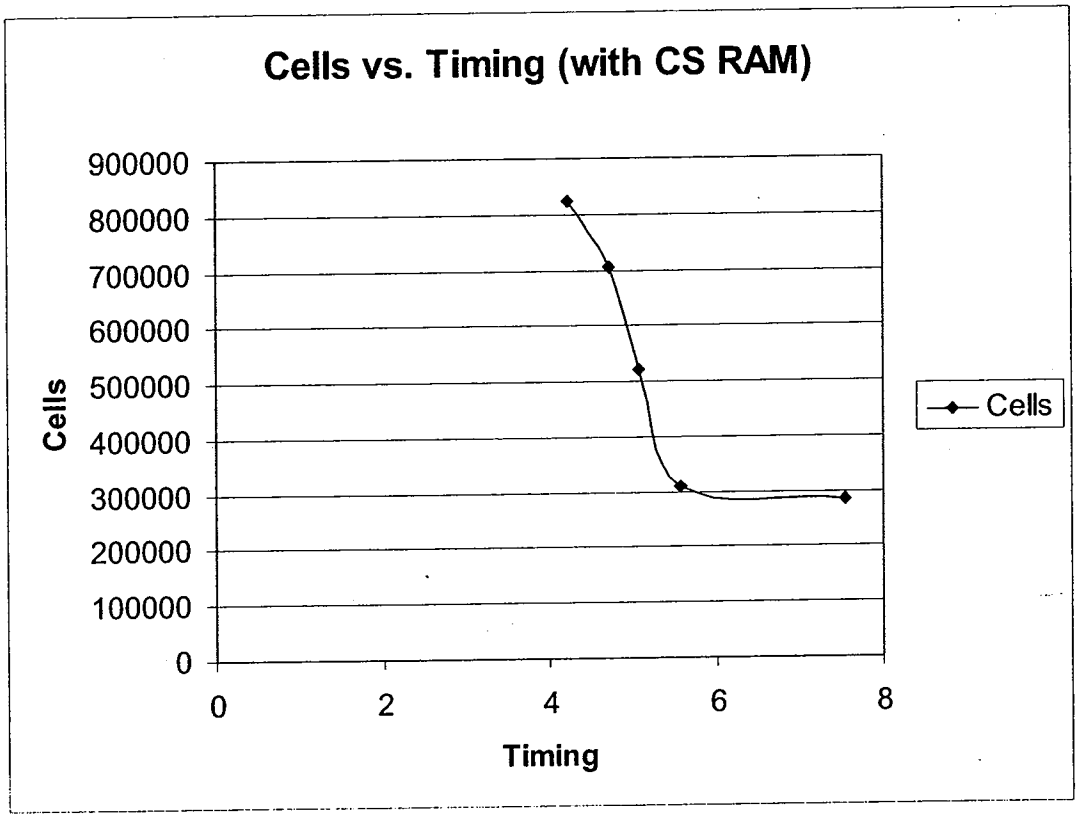

Figure 5-1: Cells vs. Timing

The core of the chip is intended to operate at 266MHz (3.75ns), but it is clear from the above graph that this frequency is not achievable with this Packet Processor architecture. Since an unrelated frequency (like 200MHz) would be very difficult to achieve due to lack of PLL resources and multiple clock domain issues, it was decided to operate at 266MHz, with each instruction taking two clock cycles. While the hardware is performing the second clock of an instruction, the unused hardware can be performing the first clock of another instruction. Thus, two independent instruction streams can be executed, providing the effect of two logical processors, achieving the performance of two 133MHz packet processors with only a little more hardware than a single packet processor.

Still, the flexibility of being able to perform many compares simultaneously, with each compare being completely flexible as to which bytes of the LHB are accessed and which constant registers were used as masks and compare values, produced an unacceptably-large design. To solve this, some comparators have been hard-wired to certain LHB locations and constant registers, while others have been restricted to smaller sets of constant registers.

6 Design Parameters

6.1 Area

The area budget for the Packet Processor is up to 380K cells including Holding Registers, Constant Data and Mask Registers, Control Store Pipeline Register, Event Registers and Flag Registers, but not including the Control Store RAM, Header Buffer or Event Queue. It is expected that the Constant Data and Mask Registers will be shared among multiple processors, reducing their cost/processor.

Outside the Packet Processor is a Header Buffer FIFO and Header Side Info Register, and an Event Queue.

The Header Buffer consists of a total of two 256-byte buffers per logical processor. The Header Side Info Register is a single 43-bit register for each logical processor.

The Event Queue consists of a total of sixteen 256-Byte buffers for the entire system, divided equally among however many Packet Processors are instantiated in the Input Processing Unit. For 2 dual-logical-processors, that would imply 4 256-Byte buffers per logical processor.

6.2 Timing

The timing budget for the Packet Processor is that it operate up to 266MHz in the IBM 0.13um process. Since a dual-logical-processor structure is to be designed, one clock cycle will be used to access the control-store RAM, and another will be used to execute the operations specified by that control word. Some of the execution cycle may be moved into the control-store access cycle to improve the timing and, therefore, the optimized gate count.

6.3 Register List

Table 6-1 below is a list of the most of the registers expected to be used in the implementation. Due to the dual-logical-processor architecture, all of these registers except the Control Register will be duplicated.

Name	# of bits
Control Register	428
LHB	2*256
Header Side register	2*43
Pipeline register	2*~250
Flags	2*16
Holding registers (4)	2*64
Event registers (5)	2*640
Pseudo-Header	2*128
Checksum accumulator	2*80
Option parser	2*38
Logical processor number	2*1
Total for 2 logical processors	**~3500**

Table 6-1: Register List

7 Timing Diagrams

7.1 Initialization Interface

The internal registers (including the Control Store RAM) are accessible via the Initialization Interface. Both read and write operations are permitted. The read and write cycles are shown below in Figure 7-1 and Figure 7-2.

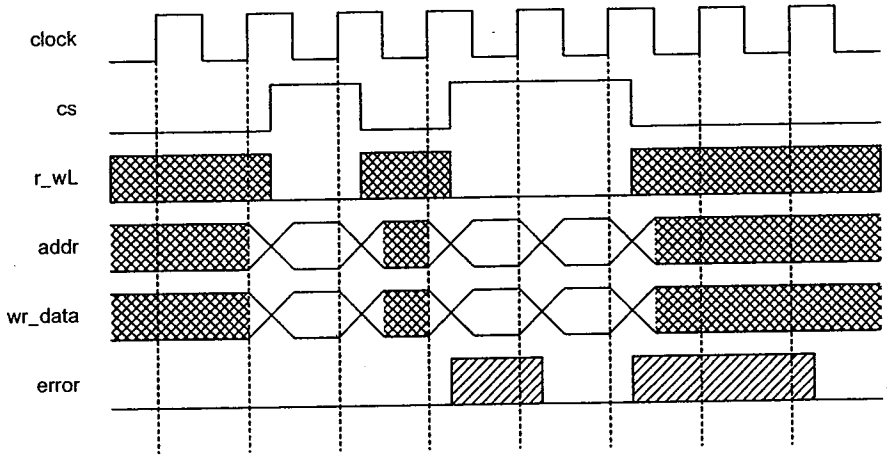

Figure 7-1: Initialization Interface Write Timing

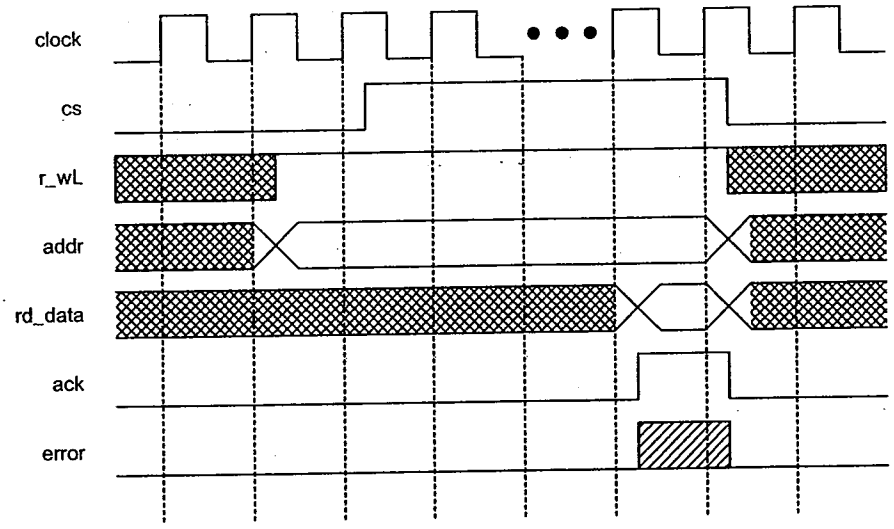

Figure 7-2: Initialization Interface Read Timing

7.2 External Configuration Interface

The External Configuration Interface is a static interface. It is required that these registers, which reside in the Input Processing Unit (see IPU HLD for details), are loaded prior to packet header processing by either of the logical processors.

7.3 Header Buffer Interface

The Header Buffer Interface has two main timing relationships: Minimum timing for "ipX_hdr_available" assertion due to assertion of "ipX_pp_done", assertion of "ipX_hdr_fifo_rd" to "ipX_hdr_data" valid. These two timing relationships are shown below in Figure 7-3.

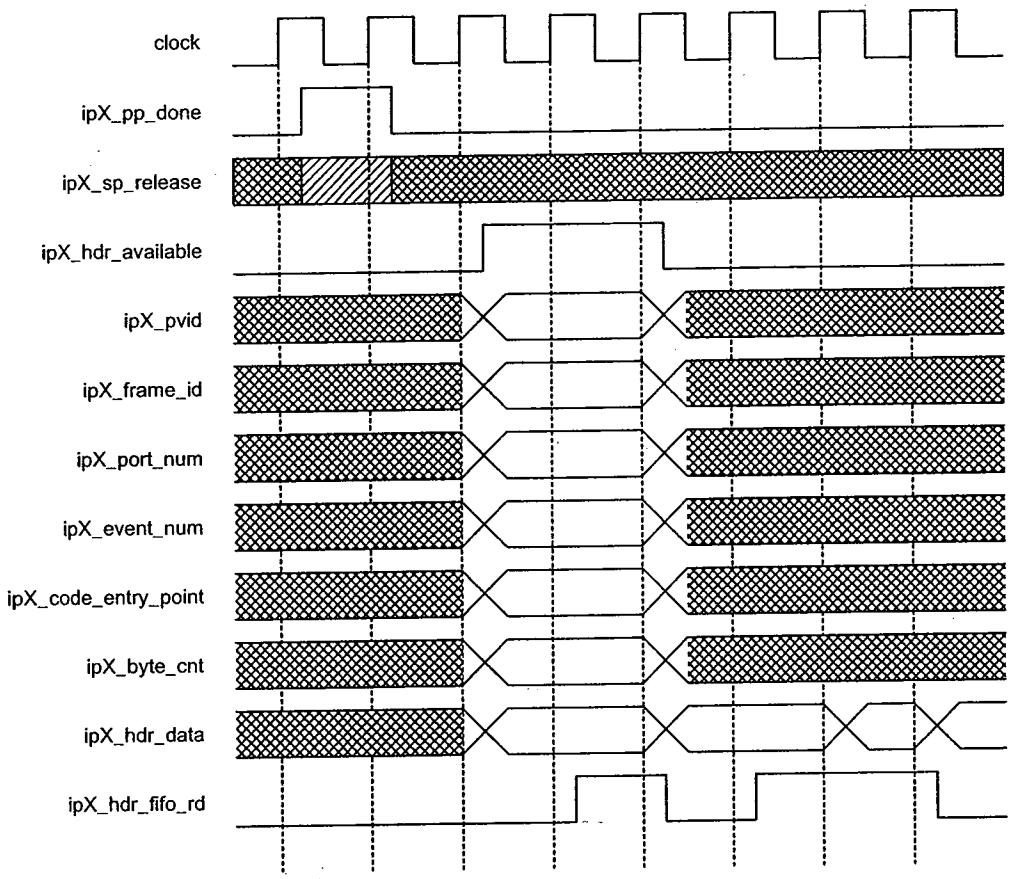

Figure 7-3: Header Buffer Interface Timing

7.4 Event Queue Interface

The Event Queue Interface has one main timing relationship: assertion of first "ipX_wvalid" after assertion of "ipX_frm_end" to "eqX_ready" valid. This timing relationship is shown below in Figure 7-4.

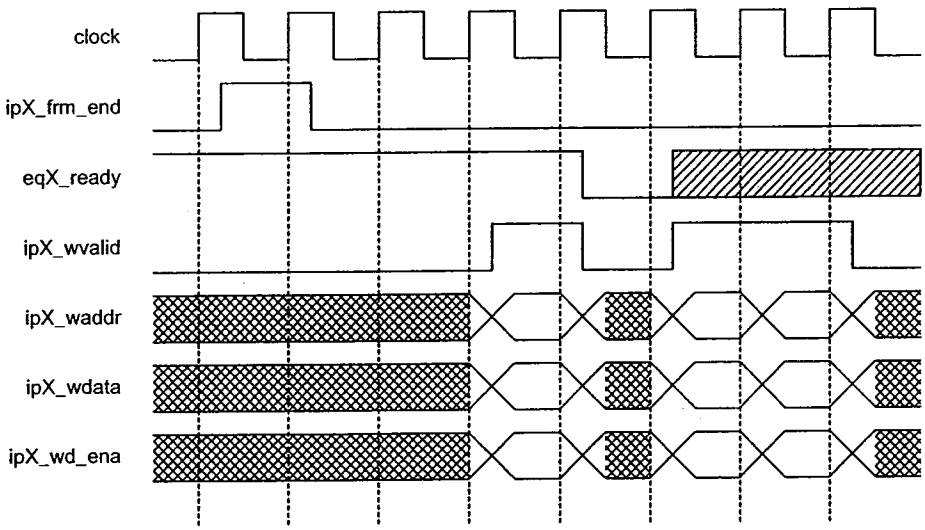

Figure 7-4: Event Queue Interface Timing

8 Implementation

The packet processor has been implemented such that packet header data alone, assuming properly written microcode, cannot cause the packet processor hardware as a whole nor any of its individual sub-units to hang.

8.1 Header Buffer (HB)

The Header Buffer (HB) resides outside the packet processor. It contains a copy of the first 256 bytes of the packet, which is referred to as the packet header. The HB is capable of buffering 2 packet headers.

8.2 Little Header Buffer (LHB)

The Little Header Buffer (LHB) acts as a 256-bit FIFO for the packet processor logic. The depth of 32-bytes is needed to guarantee that the most significant 16-bytes always contain valid data. Only these upper 16-bytes are accessible to the rest of the packet processor. The "popping" and "pushing" of the LHB are controlled by the LHB Pop Control and HB Pop Control blocks, respectively.

8.3 LHB Pop Control

The LHB Pop Control "pops" bytes from the LHB as directed by the microcode field LHB_POP. Up to 16 bytes can be "popped" from the LHB at a time. A barrel shifter is used to properly align the remaining data to the top of the LHB.

8.4 HB Pop Control

The HB Pop Control "pops" quad-words from the HB and loads them into the LHB, as space becomes available. The HB Pop Control keeps track of the number of available bytes via the microcode field LHB_POP. If at any time the number of available bytes plus the LHB_POP value meets or exceeds 16, the next quad-word is "popped" from the HB. The same barrel shifter mentioned in Section 8.3 is used to properly align the new data to the bottom of the existing data in the LHB.

8.5 Constant Data Registers

The Constant Data Registers (see Section 4.5) reside outside the packet processor (see IPU HLD) and consist of various registers that contain values to be used by the Logical and Arithmetic Units. These

registers are written during the initialization of the chip and are considered static for any and all packet processor operations.

8.6 Constant Mask Registers

The Constant Mask Registers (see Section 4.5) reside outside the packet processor (see IPU HLD) and consist of various registers that contain values to be used by the Logical and Arithmetic Units. These registers are written during the initialization of the chip and are considered static for any and all packet processor operations.

8.7 Logical Units

8.7.1 General Purpose 16-bit Comparator

The General Purpose 16-bit Comparator, shown below in Figure 8-1, can be configured (microcode field CMPx_OPERATION_SELECT, see Table 8-1) to perform one of three comparison types, which are "equal to", "greater than", and "less than". The Comparator has 3 inputs: data, mask, constant. The data can be selected (microcode field CMPx_DATA_SELECT, see Table 8-2) from any two contiguous bytes of the LHB, the Bytes Remaining register or any of the four Holding Registers. The mask can be selected (microcode field CMPx_MASK_SELECT, see Table 8-3) from any of the 7 Constant Mask Registers or all ones. The constant can be selected (microcode field CMPx_CONSTANT_SELECT, see Table 8-4) from any of the 15 Constant Data Registers or all zeros. The data is masked and compared to the constant, producing a single bit result called a "soft_compare_result". This soft_compare_result can then be stored in one of the sixteen Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch. There are a total of four Comparators in the design.

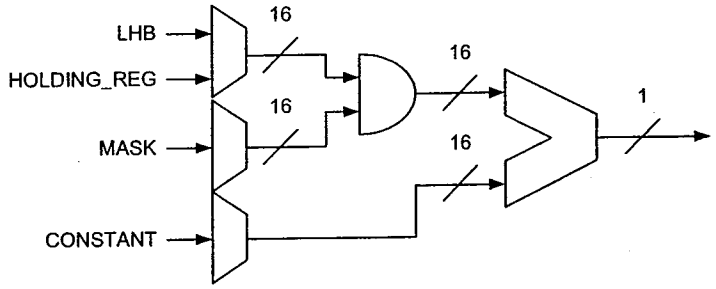

Figure 8-1: General Purpose 16-bit Comparator

CMPx_OPERATION_SELECT	Operation Type
0	Masked Data Less Than Constant?
1	Masked Data Equal To Constant?
2	Masked Data Greater Than Constant?
3	Undefined

Table 8-1: Soft Compare Operation Selection Values

CMPx Data Select	Data Field
0	{ 8'b0, LHB[127:120] }
1	LHB[127:112]
2	LHB[119:104]
3	LHB[111:96]
4	LHB[103:88]
5	LHB[95:80]
6	LHB[87:72]
7	LHB[79:64]
8	LHB[71:56]
9	LHB[63:48]
10	LHB[55:40]
11	LHB[47:32]
12	LHB[39:24]
13	LHB[31:16]
14	LHB[23:8]
15	LHB[15:0]
16	Holding Register 0
17	Holding Register 1
18	Holding Register 2
19	Holding Register 3
20	Bytes Remaining Register
21-31	Undefined

Table 8-2: Soft Compare Data Selection Values

CMPx Mask Select	Mask Field
0	Constant Mask 0
1	Constant Mask 1
2	Constant Mask 2
3	Constant Mask 3
4	Constant Mask 4
5	Constant Mask 5
6	Constant Mask 6
7	0xFFFF

Table 8-3: Soft Compare Mask Selection Values

Constant Select	CMP0 Constant Field	CMP1 Constant Field	CMP2 Constant Field	CMP3 Constant Field
0	Constant Data 0	Constant Data 3	Constant Data 6	Constant Data 8
1	Constant Data 1	Constant Data 4	Constant Data 7	Constant Data 9
2	Constant Data 2	Constant Data 5	Constant Data 8	Constant Data 10
3	Constant Data 3	Constant Data 6	Constant Data 9	Constant Data 11
4	Constant Data 4	Constant Data 7	Constant Data 10	Constant Data 12
5	Constant Data 5	Constant Data 8	Constant Data 11	Constant Data 13
6	Constant Data 6	Constant Data 9	Constant Data 12	Constant Data 14
7	0x0000	0x0000	0x0000	0x0000

Table 8-4: Soft Compare Constant Selection Values

8.7.2 Hardcoded MAC Comparators

This block contains hardcoded compares for parsing the addresses and type field of the MAC header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-5. The terms in bold are from the Constant Data or Constant Mask Registers.

Comparator Name	Compare Equation
MAC_BCAST_CMP	LHB[127:80] == 0xFFFFFFFFFFFF
MAC_MCAST_CMP	LHB[120] == 1
TMAC_UCAST_CMP	(LHB[127:80] & **TMAC_MASK**) != **EXP_TMAC_DA**
PMAC_UCAST_CMP	(LHB[127:80] & **PMAC_MASK**) != **EXP_PMAC_DA**
MAC_ARP_TYPE_CMP	LHB[31:16] == 0x0806
MAC_8023_TYPE_CMP	LHB[31:16] < 0x05DD
MAC_IPV4_TYPE_CMP	LHB[31:16] == 0x0800
MAC_IPV6_TYPE_CMP	LHB[31:16] == 0x86DD
MAC_VLAN_TYPE_CMP	LHB[31:16] == 0x8100
MAC_SAP_CMP	LHB[15:0] != 0xAAAA

Table 8-5: Hardcoded MAC Comparators

8.7.3 Hardcoded VLAN Comparators

This block contains hardcoded compares for parsing the VLAN Tag and type field of the MAC header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-6.

Comparator Name	Compare Equation
VLAN_ARP_TYPE_CMP	LHB[111:96] == 0x0806
VLAN_8023_TYPE_CMP	LHB[111:96] < 0x05DD
VLAN_IPV4_TYPE_CMP	LHB[111:96] == 0x0800
VLAN_IPV6_TYPE_CMP	LHB[111:96] == 0x86DD
VLAN_SAP_CMP	LHB[95:80] != 0xAAAA
VLAN_CFI_CMP	LHB[124] == 1

Table 8-6: Hardcoded VLAN Comparators

8.7.4 Hardcoded SNAP Comparators

This block contains hardcoded compares for parsing the SNAP field and type field of the MAC header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-7.

Comparator Name	Compare Equation
SNAP_ARP_TYPE_CMP	LHB[79:64] == 0x0806
SNAP_IPV4_TYPE_CMP	LHB[79:64] == 0x0800
SNAP_IPV6_TYPE_CMP	LHB[79:64] == 0x86DD
SNAP_UI_OUI_CMP	LHB[111:80] != 0x03000000

Table 8-7: Hardcoded SNAP Comparators

8.7.5 Hardcoded ARP Comparators

This block contains hardcoded compares for parsing a ARP header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-8. The terms in bold are from the Constant Data or Constant Mask Registers.

Comparator Name	Compare Equation
ARP_PTCL_CMP	LHB[111:96] != **EXP_PROTOCOL**
ARP_L2ALEN_CMP	LHB[95:88] != 0x06
ARP_L3ALEN_CMP	LHB[87:80] != **EXP_L3A_LEN**
ARP_OP0_CMP	LHB[79:64] != 0x0001
ARP_OP1_CMP	LHB[79:64] != 0x0002

Table 8-8: Hardcoded ARP Comparators

8.7.6 Hardcoded IPv4 Comparators

This block contains hardcoded compares for parsing an IPv4 header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-9.

Comparator Name	Compare Equation
IPV4_VERSION_CMP	LHB[127:124] != 0x4
IPV4_HDRLEN_CMP	LHB[123:120] < 0x5
IPV4_OPTION_CMP	LHB[123:120] > 0x5
IPV4_TOTLEN_CMP	LHB[111:96] < (LHB[123:120] * 4)
IPV4_ICMP_TYPE_CMP	LHB[55:48] == 0x01
IPV4_TCP_TYPE_CMP	LHB[55:48] == 0x06
IPV4_UDP_TYPE_CMP	LHB[55:48] == 0x11
IPV4_FRAGOFFSET_CMP	LHB[76:64] == 0
IPV4_FIRSTFRAG_CMP	(LHB[77] == 1) && (LHB[76:64] == 0)
IPV4_LASTFRAG_CMP	(LHB[77] == 0) && (LHB[76:64] != 0)
IPV4_FRAG_CMP	(LHB[77] == 1) || (LHB[76:64] != 0)

Table 8-9: Hardcoded IPv4 Comparators

8.7.7 Hardcoded ICMP Comparators

This block contains hardcoded compares for parsing an ICMP header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this

group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-10.

Comparator Name	Compare Equation
ICMP_HDRLEN_CMP	Holding_Register(0) < 0x0004
ICMP_DATA_CMP	Holding_Register(0) < 0x000D

Table 8-10: Hardcoded ICMP Comparators

8.7.8 Hardcoded TCP Comparators

This block contains hardcoded compares for parsing a TCP header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-11.

Comparator Name	Compare Equation
TCP_HDRLEN_CMP	LHB[31:28] < 0x5
TCP_OPTION_CMP	LHB[31:28] > 0x5
TCP_IPLEN_CMP	(LHB[31:28] * 4) > Holding_Register(0)
TCP_DATA_CMP	Holding_Register(0) == (LHB[31:28] * 4)

Table 8-11: Hardcoded TCP Comparators

8.7.9 Hardcoded UDP Comparators

This block contains hardcoded compares for parsing a UDP header. Specific bits in the LHB are compared to hardcoded constants to produce single bit results called "compare_results". These compare_results can then be stored in the Flag Registers, written to one of the five Event Registers, used by the Condition Gates, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch, provided this group has been selected via the Hardcoded Compare Results Mux. The exact hardcoded comparisons are listed below in Table 8-12.

Comparator Name	Compare Equation
UDP_HDRLEN_CMP	LHB[95:80] < 0x0008
UDP_IPLEN_CMP	LHB[95:80] > Holding_Register[0]
UDP_DATA_CMP	LHB[95:80] > 0x0008

Table 8-12: Hardcoded UDP Comparators

8.7.10 Hardcoded Compare Results Mux

This block contains the muxing of all the hardcoded compare results. At most, eleven hardcoded compare results are accessible on any given clock cycle. The hardcoded compare results are grouped as described in Section 8.7.2 through Section 8.7.8. One group also contains the Checksum Accummulator checksum validation result (ca_result_invalid).
Each input can be selected (microcode field HARD_COMPARE_GROUP_SELECT, see Table 8-13) from any of the groups. These inputs are MUXed, producing eleven single bit results called "hard_compare_results", which carry an index from 0 to 10. These hard_compare_results can then be used by the Condition Gates, Flag Registers, Event Registers, Abort Condition Gate and/or Priority Encoder.

Hard Compare Name	Group	Mux Input Index
MAC_8023_TYPE_CMP	0	0
MAC_MCAST_CMP	0	1
TMAC_UCAST_CMP	0	2
PMAC_UCAST_CMP	0	3
MAC_IPV4_TYPE_CMP	0	4
MAC_ARP_TYPE_CMP	0	5
MAC_VLAN_TYPE_CMP	0	6
MAC_SAP_CMP	0	7
MAC_IPV6_TYPE_CMP	0	8
MAC_BCAST_CMP	0	9
VLAN_8023_TYPE_CMP	1	0
VLAN_IPV4_TYPE_CMP	1	1
VLAN_ARP_TYPE_CMP	1	2
VLAN_SAP_CMP	1	3
VLAN_IPV6_TYPE_CMP	1	4
VLAN_CFI_CMP	1	5
SNAP_UI_OUI_CMP	2	0
SNAP_IPV4_TYPE_CMP	2	1
SNAP_ARP_TYPE_CMP	2	2
SNAP_IPV6_TYPE_CMP	2	3
ARP_PTCL_CMP	3	0
ARP_L2ALEN_CMP	3	1
ARP_L3ALEN_CMP	3	2
ARP_OP0_CMP	3	3
ARP_OP1_CMP	3	4
IPV4_VERSION_CMP	4	0
IPV4_HDRLEN_CMP	4	1
IPV4_OPTION_CMP	4	2
IPV4_TOTLEN_CMP	4	3
IPV4_UDP_TYPE_CMP	4	4
IPV4_LASTFRAG_CMP	4	5
IPV4_FRAG_CMP	4	6
IPV4_ICMP_TYPE_CMP	4	7
IPV4_TCP_TYPE_CMP	4	8
IPV4_FIRSTFRAG_CMP	4	9
IPV4_FRAGOFFSET_CMP	4	10
UDP_HDRLEN_CMP	5	1
UDP_DATA_CMP	5	2
UDP_IPLEN_CMP	5	3
Flag Register 9	5	4
Flag Register 10	5	5

Flag Register 11	5	6
Flag Register 12	5	7
Flag Register 13	5	8
Flag Register 14	5	9
Flag Register 15	5	10
ICMP_DATA_CMP	6	0
ICMP_HDRLEN_CMP	6	1
Flag Register 0	6	2
Flag Register 1	6	3
Flag Register 2	6	4
Flag Register 3	6	5
Flag Register 4	6	6
Flag Register 5	6	7
Flag Register 6	6	8
Flag Register 7	6	9
Flag Register 8	6	10
CA_RESULT_INVALID	7	0
TCP_HDRLEN_CMP	7	1
TCP_OPTION_CMP	7	2
TCP_IPLEN_CMP	7	3
TCP_DATA_CMP	7	4

Table 8-13: Hardcoded Compare Results Mux Input Map

8.8 Condition Gate Units

8.8.1 Four Input AND gate

The Four Input Condition AND Gate, shown below in Figure 8-2, can AND up to four compare_results. Each input can be selected (microcode field ANDx_INPUTx_DATA_SELECT, see Table 8-14) from any of the compare_results (soft and hard). Each input can also be inverted (microcode field ANDx_INVERSION_SELECT) to obtain the compliment of the selected compare_result. The four inputs are ANDed, producing a single bit result called a "cgate_result". This cgate_result can then be stored in one of the sixteen Flag Registers, written to one of the five Event Registers, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch. There are a total of four Condition AND Gates in the design.

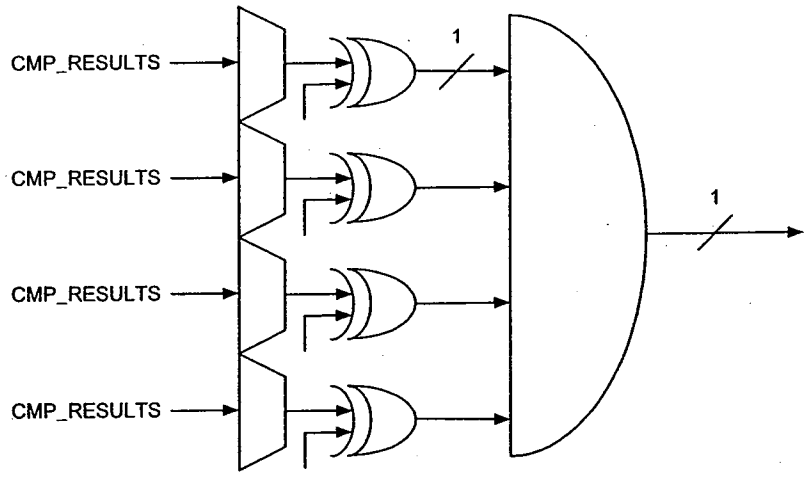

Figure 8-2: Condition AND Gate

ANDx Input Select	Input Field
0	Soft Compare Result 0
1	Soft Compare Result 1
2	Soft Compare Result 2
3	Soft Compare Result 3
4	Hard Compare Result 0
5	Hard Compare Result 1
6	Hard Compare Result 2
7	Hard Compare Result 3
8	Hard Compare Result 4
9	Hard Compare Result 5
10	Hard Compare Result 6
11	Hard Compare Result 7
12	Hard Compare Result 8
13	Hard Compare Result 9
14	Hard Compare Result 10
15	Logic 1

Table 8-14: Condition AND Gate Selection Values

8.8.2 Four input OR gate

The Four Input Condition OR Gate, shown below in Figure 8-3, can OR up to four compare_results. Each input can be selected (microcode field ORx_INPUTx_DATA_SELECT, see Table 8-15) from any of the compare_results (soft and hard). Each input can also be inverted (microcode field ORx_INVERSION_SELECT) to obtain the compliment of the selected compare_result. The four inputs are ORed, producing a single bit result called a "cgate_result". This cgate_result can then be stored in one of the sixteen Flag Registers, written to one of the five Event Registers, used by the Abort Condition Gate and/or used by the Priority Encoder to conditionally branch. There are a total of two Condition OR Gates in the design.

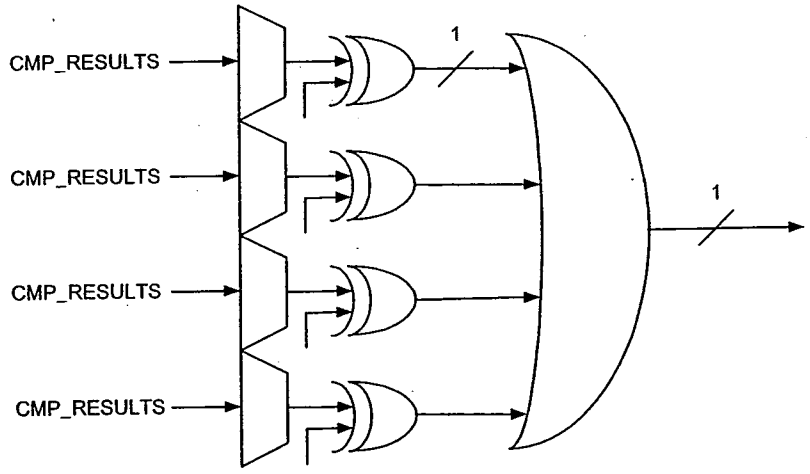

Figure 8-3: Condition OR Gate

ORx Input Select	Input Field
0	Soft Compare Result 0
1	Soft Compare Result 1
2	Soft Compare Result 2
3	Soft Compare Result 3
4	Hard Compare Result 0
5	Hard Compare Result 1
6	Hard Compare Result 2
7	Hard Compare Result 3
8	Hard Compare Result 4
9	Hard Compare Result 5
10	Hard Compare Result 6
11	Hard Compare Result 7
12	Hard Compare Result 8
13	Hard Compare Result 9
14	Hard Compare Result 10
15	Logic 0

Table 8-15: Condition OR Gate Selection Values

8.9 Flag Register

The Flag Register, shown below in Figure 8-4, can store a compare_result or a cgate_result. The Flag Register has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_FLAG_REGx. The data can be selected (microcode field FLAG_REGx_DATA_SELECT, see Table 8-16) from a subset of all the compare_results and cgate_results. There a total of 16 Flag Registers in the design. Each Flag Register has a different data selection subset such that all the compare_results and cgate_results can be stored.

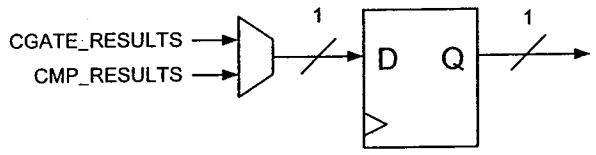

Figure 8-4: Flag Register

Flag Register	Input Select = 0	Input Select = 1	Input Select = 2	Input Select = 3
0	Hard Compare Result 0	Soft Compare Result 0	AND Result 0	OR Result 1
1	Hard Compare Result 1	Soft Compare Result 1	AND Result 0	OR Result 0
2	Hard Compare Result 3	Soft Compare Result 3	AND Result 1	AND Result 2
3	Hard Compare Result 6	Soft Compare Result 2	AND Result 3	OR Result 1
4	Hard Compare Result 0	Hard Compare Result 4	Soft Compare Result 3	AND Result 2
5	Hard Compare Result 1	Hard Compare Result 6	Soft Compare Result 1	AND Result 3
6	Hard Compare Result 9	Soft Compare Result 2	AND Result 1	OR Result 0
7	Hard Compare Result 2	Hard Compare Result 5	Soft Compare Result 0	AND Result 0
8	Hard Compare Result 3	Soft Compare Result 2	Soft Compare Result 3	OR Result 1
9	Soft Compare Result 0	Soft Compare Result 2	AND Result 1	AND Result 3
10	Hard Compare Result 7	Soft Compare Result 1	AND Result 0	OR Result 0
11	Hard Compare Result 8	Soft Compare Result 2	AND Result 1	AND Result 2
12	Hard Compare Result 9	Soft Compare Result 3	AND Result 2	OR Result 1
13	Soft Compare Result 0	Soft Compare Result 2	AND Result 3	OR Result 0
14	Soft Compare Result 1	Soft Compare Result 3	AND Result 0	AND Result 1
15	Hard Compare Result 10	Soft Compare Result 0	AND Result 3	OR Result 1

Table 8-16: Flag Register Input Subsets

8.10 Abort Condition Unit

The Four Input Abort Condition OR Gate, shown below in Figure 8-5, can OR up to four compare_results, cgate_results and/or flag registers. Each of the four inputs can be selected (microcode field ABORT_INPUTx_DATA_SELECT, see Table 8-17) from a subset of all the compare_results, cgate_results and flag registers. The four inputs are ORed, producing a single bit result called an "abort_result". This abort_result can then be used by the Priority Encoder to conditionally branch.

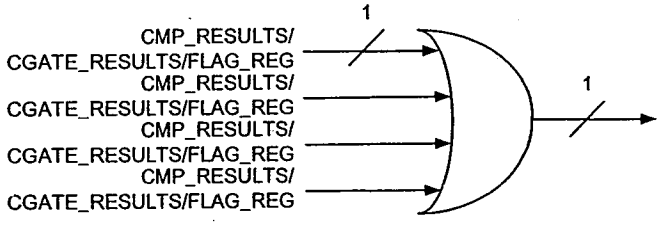

Figure 8-5: Abort OR Gate

Input Select	Input 0	Input 1	Input 2	Input 3
0	Hard Compare Result 0	Hard Compare Result 4	Hard Compare Result 3	Hard Compare Result 5
1	Hard Compare Result 1	Soft Compare Result 2	Soft Compare Result 0	Soft Compare Result 1
2	Soft Compare Result 0	Soft Compare Result 3	Soft Compare Result 3	Soft Compare Result 2
3	Soft Compare Result 1	AND Result 2	AND Result 1	AND Result 2
4	AND Result 0	AND Result 3	AND Result 3	OR Result 1
5	AND Result 1	OR Result 1	OR Result 0	Flag Register 9
6	OR Result 0	Flag Register 7	Flag Register 8	Flag Register 15
7	Logic 0	Logic 0	Logic 0	Logic 0

Table 8-17: Abort OR Gate Input Subsets

8.11 Arithmetic Units

8.11.1 General Purpose 16-bit Shift Left Adder/Subtractor

The General Purpose 16-bit shift left adder/subtractor Arithmetic Unit (AU0), shown below in Figure 8-6, can be configured (microcode field AU0_OPERATION_SELECT, see Table 8-18) to "multiply by 4" and perform one of two operation types, which are "addition" and "subtraction". The AU0 has 3 inputs: data, mask, constant. The data can be selected (microcode field AU0_DATA_SELECT, see Table 8-19) from any two contiguous bytes of the LHB, any of the four Holding Registers, the Bytes Remaining register or the 16-bit microcode field IMMEDIATE_DATA. The mask (microcode field AU0_MASK_SELECT, see Table 8-20) can be selected from any of the 7 Constant Mask Registers or all ones. The constant can be selected (microcode field AU0_CONSTANT_SELECT, see Table 8-21) from 12 of the 15 Constant Data Registers or any of the 4 Holding Registers. The data is masked and possibly multiplied by 4 (this is really just a shift left by 2 on the masked data) and then added to or subtracted from the constant, producing a 16-bit result called an "au0_result". This au0_result can then be stored in one of the 4 Holding Registers and/or written to one of the five Event Registers.

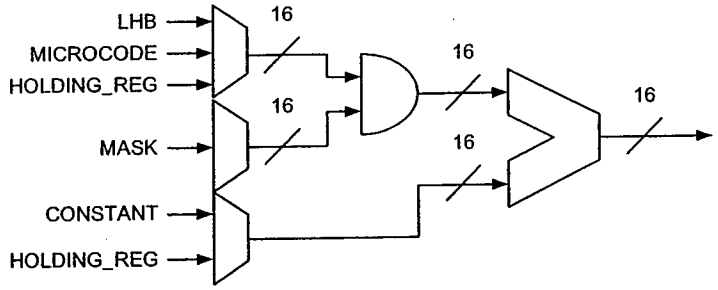

Figure 8-6: General Purpose Shift Left Add/Sub

AU0 Operation Select	Operation Type
0	Masked Data Plus Constant
1	Masked Data Minus Constant
2	Shifted Masked Data Plus Constant
3	Shifted Masked Data Minus Constant

Table 8-18: Arithmetic Unit 0 Operation Selection Values

AU0 Data Select	Data Field
0	{ 8'b0, LHB[127:120] }
1	LHB[127:112]
2	LHB[119:104]
3	LHB[111:96]
4	LHB[103:88]
5	LHB[95:80]
6	LHB[87:72]
7	LHB[79:64]
8	LHB[71:56]
9	LHB[63:48]
10	LHB[55:40]
11	LHB[47:32]
12	LHB[39:24]
13	LHB[31:16]
14	LHB[23:8]
15	LHB[15:0]
16	Holding Register 0
17	Holding Register 1
18	Holding Register 2
19	Holding Register 3
20	Bytes Remaining Register
21	Microcode Immediate Data Field
22-31	Undefined

Table 8-19: Arithmetic Unit 0 Data Selection Values

AU0 Mask Select	Mask Field
0	Constant Mask 0
1	Constant Mask 1
2	Constant Mask 2
3	Constant Mask 3
4	Constant Mask 4
5	Constant Mask 5
6	Constant Mask 6
7	0xFFFF

Table 8-20: Arithmetic Unit 0 Mask Selection Values

AU0 Constant Select	Constant Field
0	Constant Data 0
1	Constant Data 1
2	Constant Data 2
3	Constant Data 3
4	Constant Data 4
5	Constant Data 5
6	Constant Data 6
7	Constant Data 7
8	Constant Data 8
9	Constant Data 9
10	Constant Data 10
11	Constant Data 11
12	Holding Register 0
13	Holding Register 1
14	Holding Register 2
15	Holding Register 3

Table 8-21: Arithmetic Unit 0 Constant Selection Values

8.11.2 General Purpose 16-bit Shift Right Adder/Subtractor

The General Purpose 16-bit shift right adder/subtractor Arithmetic Unit (AU1), shown below in Figure 8-7, can be configured (microcode field AU1_OPERATION_SELECT, see Table 8-22) to "divide by 16" and perform one of two operation types, which are "addition" and "subtraction". The AU1 has 3 inputs: data, mask, constant. The data can be selected (microcode field AU1_DATA_SELECT, see Table 8-23) from any two contiguous bytes of the LHB, any of the four Holding Registers, the Bytes Remaining register or the 16-bit microcode field IMMEDIATE_DATA. The mask (microcode field AU1_MASK_SELECT, see Table 8-24) can be selected from any of the 7 Constant Mask Registers or all ones. The constant can be selected (microcode field AU1_CONSTANT_SELECT, see Table 8-25) from 12 of the 15 Constant Data Registers or any of the 4 Holding Registers. The data is masked and possibly divided by 16 (this is really just a shift right by 4 on the masked data) and then added to or subtracted from the constant, producing a 16-bit result called an "au1_result". This au1_result can then be stored in one of the 4 Holding Registers and/or written to one of the five Event Registers.

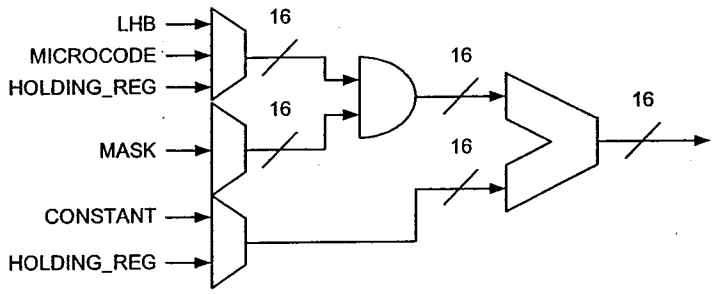

Figure 8-7: General Purpose Shift Right Add/Sub

AU1 Operation Select	Operation Type
0	Masked Data Plus Constant
1	Masked Data Minus Constant
2	Shifted Masked Data Plus Constant
3	Shifted Masked Data Minus Constant

Table 8-22: Arithmetic Unit 1 Operation Selection Values

AU1 Data Select	Data Field
0	{ 8'b0, LHB[127:120] }
1	LHB[127:112]
2	LHB[119:104]
3	LHB[111:96]
4	LHB[103:88]
5	LHB[95:80]
6	LHB[87:72]
7	LHB[79:64]
8	LHB[71:56]
9	LHB[63:48]
10	LHB[55:40]
11	LHB[47:32]
12	LHB[39:24]
13	LHB[31:16]
14	LHB[23:8]
15	LHB[15:0]
16	Holding Register 0
17	Holding Register 1
18	Holding Register 2
19	Holding Register 3
20	Bytes Remaining Register
21	Microcode Immediate Data Field
22-31	Undefined

Table 8-23: Arithmetic Unit 1 Data Selection Values

AU1 Mask Select	Mask Field
0	Constant Mask 0
1	Constant Mask 1
2	Constant Mask 2
3	Constant Mask 3
4	Constant Mask 4
5	Constant Mask 5
6	Constant Mask 6
7	0xFFFF

Table 8-24: Arithmetic Unit 1 Mask Selection Values

AU1 Constant Select	Constant Field
0	Constant Data 3
1	Constant Data 4
2	Constant Data 5
3	Constant Data 6
4	Constant Data 7
5	Constant Data 8
6	Constant Data 9
7	Constant Data 10
8	Constant Data 11
9	Constant Data 12
10	Constant Data 13
11	Constant Data 14
12	Holding Register 0
13	Holding Register 1
14	Holding Register 2
15	Holding Register 3

Table 8-25: Arithmetic Unit 1 Constant Selection Values

8.12 Holding Register

The 16-bit Holding Register, shown below in Figure 8-8, can store a 16-bit value for future use. The Holding Register has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_HOLDING_REGx. The data can be selected (microcode field HOLDING_REGx_DATA_SELECT, see Table 8-26) from either au0_result or au1_result. The output of the Holding Register can be used by the Comparators (hard or soft) as well as the arithmetic units. There are a total of four Holding Registers in the design.

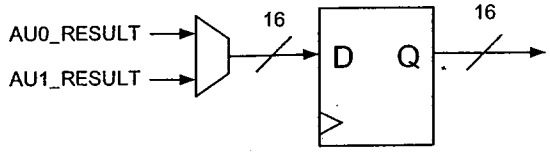

Figure 8-8: Holding Register

Data Select	Input Data Field
0	Arithmetic Unit 0 Result
1	Arithmetic Unit 1 Result

Table 8-26: Holding Register Input Selection Values

8.13 Event Register

The 128-bit Event Register is a temporary register that is used to create portions of the event structure that are not directly copied from the LHB. There are a total of five Event Registers in the design, each of which is specifically assigned to a protocol type or types. Once all the necessary portions of the event structure have been created, the appropriate event registers are copied to the Event Queue. The exact assignments of the Event Registers are listed below in Table 8-27.

Event Register	Assigned Protocol Type
0	Control 0 – appears in all event structures, regardless of the protocol type
1	Control 1 – appears in all event structures, regardless of the protocol type
2	MAC 0 – contains MAC specific fields, such as VLAN Tag and the SPI4 port number
3	ARP_IPv4 – contains ARP & IPv4 specific fields
4	ICMP_UDP_TCP – contains ICMP, UDP & TCP specific fields

Table 8-27: Event Register Assignments

8.13.1 Event Flags

The most significant 16 bits of Event Registers 2 thru 4 are reserved for Event Flags. Each Event Flag has 3 inputs: data, load enable, set enable. Only the Event Flags of a single Event Register can be loaded (microcode field LOAD_EVENT_FLAGx) or set (microcode field SET_EVENT_FLAG) on any given clock cycle as directed by the microcode field ER_EVENT_FLAG_ADDRESS, see Table 8-28. The Event Flags and Flag Registers share the same input muxes. Therefore, the data for a load can be selected from the same sixteen subsets described previously for the Flag Registers and listed in Table 8-16. The data for a set is taken from the microcode field IMMEDIATE_DATA. If a bit is set in the microcode field IMMEDIATE_DATA, the corresponding Event Flag is also set. If a bit is not set in the microcode field IMMEDIATE_DATA, the corresponding Event Flag retains its current value.

Address	Event Register
0	Event Register 2 (MAC0)
1	Event Register 3 (ARP/IPv4)
2	Event Register 4 (ICMP/TCP/UDP)
3	Undefined

Table 8-28: Event Register Selection Values

8.13.2 Create-Flow Flag

The create-flow flag, shown below in Figure 8-9, is the most significant bit of Event Register 0. The create-flow flag has 4 inputs: load enable, data, mask, constant. The load enable is controlled by the microcode field LOAD_CREATE_FLOW_FLAG. The data is extracted from LHB[21:16]. The mask is taken from the Constant Mask Register CF_MASK. The constant is taken from the Constant Data Register EXP_TCP_FLAGS. The data is masked and compared to the constant, producing a single bit result called a "cf_result". This cf_result can then be stored in bit 127 of Event Register 0.

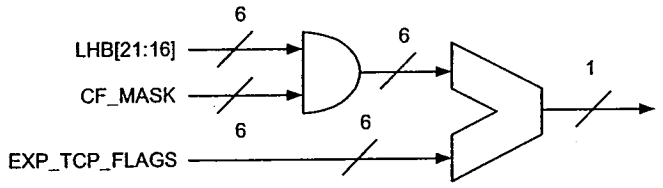

Figure 8-9: Create-Flow Flag

8.13.3 Event Type Field

The Event Type field is a 6-bit field contained in the most significant quad-word of the Control Event Structure. The Event Type field has 2 inputs: load enable, data. The load enable is controlled by the

microcode field LOAD_EVENT_TYPE. The data is extracted from the microcode field IMMEDIATE_DATA[13:8] and written into bits 121-116 of Event Register 0.

8.13.4 Flow Key Field

The Flow Key field is a 116-bit field that spans both quad-words of the Control Event Structure. The Flow Key field contains various sub-fields, which are: Receive Interface (12 bits), Layer 4 Source Port (16 bits), Layer 4 Destination Port (16 bits), Layer 3 Source Address, Layer 3 Destination Address, Layer 3 Protocol. The exact Flow Key bit assignments are listed below in Table 8-29.

Bits	Description
115:112	Receive Interface[11:8]
111:96	Layer 4 Source Port
95:80	Layer 4 Destination Port
79:48	Layer 3 Source Address
47:16	Layer 3 Destination Address
15:8	Receive Interface[7:0]
7:0	Layer 3 Protocol

Table 8-29: Flow Key Bit Map

The Flow Key has a load enable (microcode field LOAD_FLOW_KEY_x) for each of the sub-fields. The Flow Key is constructed as various portions of the packet are processed. When directed by the microcode, data is extracted from specific bits in the LHB and written into the appropriate Event Register, with one exception. The Receive Interface has 3 possible sources, none of which are the LHB. The Receive Interface source is selected by the RCVIFSRCSEL bits of the Control Register. If RCVIFSRCSEL = 00, the source is all zeros. If RCVIFSRCSEL = 01, the source is the value contained in the VLAN ID sub-field of the VLAN Tag field of Event Register 2. If RCVIFSRCSEL = 10, the source is the IPU ID input pin value concatenated with the value contained in the SPI4 Port Number field of Event Register 2. For all scenarios, the four most significant bits of data are written into bits 99-96 of Event Register 0 and the remaining eight bits of data are written into bits 127-120 of Event Register 1.
Data for the Layer 4 Source Port and the Layer 4 Destination Port is extracted as a single block from LHB[127:96], ANDed with Constant Mask Register LPx_FLOW_KEY_SPDP_MASK and written into bits 95-64 of Event Register 0. Data for the Layer 3 Source Address is extracted from LHB[63:32], ANDed with Constant Mask Register LPx_FLOW_KEY_SA_MASK and written into bits 63-32 of Event Register 0. Data for the Layer 3 Destination Address is extracted from LHB[31:0], ANDed with Constant Mask Register LPx_FLOW_KEY_DA_MASK and written into bits 31-0 of Event Register 0. Data for the Layer 3 Protocol is extracted from LHB[87:80], ANDed with Constant Mask Register LPx_FLOW_KEY_PTCL_MASK. The result is then ORed with Constant Mask Register LPx_FLOW_KEY_PTCL_FORCE and written into bits 119-112 of Event Register 1.

8.13.5 Event Size Field

The Event Size field is a 5-bit field contained in the least significant quad-word of the Control Event Structure. The Event Size field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_EVENT_SIZE. The data is extracted from the microcode field IMMEDIATE_DATA[12:8] and written into bits 108-104 of Event Register 1.

Firmware Note: The event size must be written by the microcode. For ARP and TCP (with and without options) the event structure is a fixed size. For ICMP, UDP and unrecognized protocols, as much frame data as possible is to be copied into the event structure and the event size is set to 16 regardless of how much data was actually copied. It is up to the protocol core to figure out how much useful data is there.

8.13.6 Event Subcode Field

The Event Subcode field is an 8-bit field contained in the least significant quad-word of the Control Event Structure. The Event Subcode field has 2 inputs: load enable, data. The load enable is controlled by the

microcode field LOAD_EVENT_SUBCODE. The data is extracted from the microcode field IMMEDIATE_DATA[7:0] and written into bits 103-96 of Event Register 1.

8.13.7 SP DATA Field

The SP Data field is a 1-bit field contained in the least significant quad-word of the Control Event Structure. The SP Data field has 2 inputs: load enable, data. The load enable is controlled by the microcode field DONE. The data source is controlled by FORCE_SP_DATA. If FORCE_SP_DATA is '1', the data is the value present on SP_DATA_VAL. If FORCE_SP_DATA is '0', the data is the compliament of the microcode field SP_RELEASE. The data is written into bit 81 of Event Register 1.

8.13.8 Payload Scratch Offset Field

The Payload Scratch Offset field is an 8-bit field contained in the least significant quad-word of the Control Event Structure. The Payload Scratch Offset field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_PS_OFFSET. The data is copied from the internal Bytes Popped Register, which accumulates the number of bytes that have been "popped" from the LHB, and written into bits 63-56 of Event Register 1.

8.13.9 Frame ID Field

The Frame ID field is an 8-bit field contained in the least significant quad-word of the Control Event Structure. The Frame ID field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_IPU_SPI4_FRAME_ID. The data is extracted from the header side information (FRAME_ID) and written into bits 31-24 of Event Register 1.

8.13.10 Running Checksum Field

The Running Checksum field is a 16-bit field contained in the least significant quad-word of the Control Event Structure. The Running Checksum field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_RUNNING_CSUM. The data has 2 possible sources. The data is copied from the Checksum Accumulator unit, which calculates a ones compliment checksum as directed by the microcode, and written into bits 15-0 of Event Register 1.

8.13.11 VLAN Tag Field

The VLAN Tag field is a 16-bit field contained in the most significant quad-word of the MAC Event Structure. The VLAN Tag field contains 3 sub-fields, which are: Priority (3 bits), CFI (1 bit), VLAN ID (12 bits). The exact Flow Key bit assignments are listed below in Table 8-30.

Bits	Description
15:13	Priority
12	Canonical Format Indicator (CFI)
11:0	VLAN ID

Table 8-30: VLAN Tag Bit Map

The VLAN Tag field has a single load enable for the 3 sub-fields. The load enable is controlled by the microcode field LOAD_VLAN_TAG. The data for each sub-field has 2 possible sources. The data source is determined by the microcode field VLAN_TAG_DATA_SELECT, see Table 8-31. The data is extracted based on whether or not the packet is VLAN tagged. If the packet is VLAN tagged, the data is copied from LHB[127:112] into bits 111-96 of Event Register 2. If the packet is not VLAN tagged, the data for the VLAN ID sub-field is extracted from the header side information (PVID) and written into bits 107-96 of Event Register 2. The remaining sub-fields are to be all zeros.

Data Select	Data Input Field
0	PVID
1	LHB[127:112]

Table 8-31: VLAN Tag Data Selection Values

8.13.12 Fabric Header Length Field

The Fabric Header Length field is a 5-bit field contained in the most significant quad-word of the MAC Event Structure. The Fabric Header Length field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_SF_HEADER_LENGTH. The data is extracted from the microcode field LHB_POP and written into bits 92-88 of Event Register 2.

8.13.13 IPU Number Field

The IPU Number field is a 1-bit field contained in the most significant quad-word of the MAC Event Structure. The IPU Number field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_IPU_SPI4_FRAME_ID. The data is copied from the External Configuration Interface (IPU_ID) into bit 72 of Event Register 2.

8.13.14 SPI4 Port Number Field

The SPI4 Port Number field is an 8-bit field contained in the most significant quad-word of the MAC Event Structure. The SPI4 Port Number field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_IPU_SPI4_FRAME_ID. The data is extracted from the header side information (PORT_NUM) and written into bits 71-64 of Event Register 2.

8.13.15 L3 Header Scratch Offset Field

The L3 Header Scratch Offset field is an 8-bit field contained in the most significant quad-word of the MAC Event Structure. The L3 Header Scratch Offset field has 2 inputs: load enable, data. The load enable is controlled by the microcode field LOAD_L3_HEADER_OFFSET. The data is copied from the internal Bytes Popped Register, which accumulates the number of bytes that have been "popped" from the LHB, and written into bits 87-80 of Event Register 2.

8.13.16 Option Length Field

The Option Length field is a 4-bit field contained in the IPv4 Event Structure and the TCP Event Structure. These fields are calculated and written separately. The Option Length field has 3 inputs: type, load enable, data. The type is controlled by the microcode field OP_OPTION_TYPE. The load enable is controlled by the microcode field LOAD_IP_TCP_OPTION_LENGTH. The data is copied from the output of one of the two Arithmetic Units as directed by the microcode (IP_TCP_OPTION_LENGTH_DATA_SELECT, see Table 8-32), and written into bits 107-104 of the appropriate Event Register (3 for IPv4, 4 for TCP).

Data Select	Input Data Field
0	Arithmetic Unit 0 Result
1	Arithmetic Unit 1 Result

Table 8-32: Option Length Data Selection Values

8.13.17 Option Offset Field

The Option Offset field is an 8-bit field contained in the IPv4 Event Structure and the TCP Event Structure. These fields are calculated and written separately. The Option Offset field has 3 inputs: type, load enable, data. The type is controlled by the microcode field OP_OPTION_TYPE. The load enable is controlled by the microcode field LOAD_IP_TCP_OPTION_OFFSET. The data is extracted from the microcode field IMMEDIATE_DATA[7:0] and written into bits 103-96 of the appropriate Event Register (3 for IPv4, 4 for TCP).

8.13.18 Option Byte Offset Fields

The Option Byte Offset fields are a group of bytes contained in the IPv4 Event Structure and the TCP Event Structure. The IPv4 Event Structure has a group of 3 bytes, while the TCP Event Structure has a group of 10 bytes. The exact Option Byte Offset bit assignments are listed below in Table 8-33.

Event Register	Bits	Option Byte Offset Field	Corresponding Option Type Bit Map
3	95:88	IPv4 Option Byte Offset 0	ip_option_types[7:0]
3	87:80	IPv4 Option Byte Offset 1	ip_option_types[15:8]
3	79:72	IPv4 Option Byte Offset 2	ip_option_types[23:16]
4	95:88	TCP Option Byte Offset 0	tcp_option_types[7:0]
4	87:80	TCP Option Byte Offset 1	tcp_option_types[15:8]
4	79:72	TCP Option Byte Offset 2	tcp_option_types[23:16]
4	71:64	TCP Option Byte Offset 3	tcp_option_types[31:24]
4	63:56	TCP Option Byte Offset 4	tcp_option_types[39:32]
4	55:48	TCP Option Byte Offset 5	tcp_option_types[47:40]
4	47:40	TCP Option Byte Offset 6	tcp_option_types[55:48]
4	39:32	TCP Option Byte Offset 7	tcp_option_types[63:56]
4	31:24	TCP Option Byte Offset 8	tcp_option_types[71:64]
4	23:16	TCP Option Byte Offset 9	tcp_option_types[79:72]

Table 8-33: Option Byte Offset Bit Map

These fields are calculated and written separately. Each of the Option Byte Offset fields have 3 inputs: type, load enable, data. The load enable is controlled by the Option Parsing Unit. The data is written into the appropriate Event Register (3 for IPv4, 4 for TCP) by the Option Parsing Unit, as directed by the microcode field OP_OPTION_TYPE.

8.13.19 SACK Blocks Field

The SACK Blocks field is a 4-bit field contained in the TCP Event Structure. The SACK Blocks field has 2 inputs: load enable, data. The load enable is controlled by the Option Parsing Unit. The data is written into bits 3-0 of Event Register 4 by the Option Parsing unit, as directed by the microcode.

8.14 Option Parsing Unit

The Option Parsing Unit can be configured (microcode field OP_OPTION_TYPE, see Table 8-34) to parse either IPv4 or TCP options. Once enabled (microcode field OP_ENABLE), the option parsing unit compares the four most significant bytes of the LHB to either 3 possible IP option types or 10 possible TCP option types. These option types are defined by Constant Data Registers ip_option_types and tcp_option_types, respectively. The correlation between the ip_option_types and tcp_option_types to their respective Option Byte Offset fields can be found in Table 8-33. The Option Parsing Unit parses, at most, one option (up to 16 bytes) or up to four NOPs. If a match is found the byte offset from the start of the header to the Length byte of the option is written into the appropriate Event Register (3 for IPv4, 4 for TCP). The Option Parsing Unit assumes that the least significant byte of tcp_option_types specifies the SACK option. If the Type field matches this byte, the number of SACK blocks is calculated from the Length field and written into Event Register 4. The Option Parsing Unit, once enabled, also controls popping the LHB, checksum accumulation, conditional branching and writing to the Event Queue with one exception. The microcode field EQ_WRITE_ENABLE must be set to 0x0 for the entire time that the Option Parsing Unit is enabled. All this is necessary to guarantee that it can parse options in any order without hanging. When the Option Parsing Unit encounters an option type that is not one of the possible matches, the "unknown_option" flag of the appropriate Event Register is set. If the Option Parsing Unit encounters an EOL, it will pop the remaining option bytes from the LHB before deasserting the OP_NOT_DONE flag. If the Option Parsing Unit encounters an option length of less than 2 bytes or greater than the number of remaining option bytes to process for an option type that is not NOP nor EOL, it will set the "op_parse_error" flag of the appropriate Event Register and it will assert the OP_ABORT flag, causing an abort branch.

During the same clock in which it is enabled, the Option Parsing Unit loads its internal registers with certain values such that it can be ready to parse on the following cycle. To this end, four of the internal registers require calculated values the clock cycle in which it is enabled. The internal register op_byte_offset_reg requires the byte offset from the beginning of the header (IPv4 or TCP) to the first byte where the options are to be written (microcode field IMMEDIATE_DATA[7:0]). The internal register op_eq_wr_addr_reg requires the Event Queue starting address for where the options are to be copied (microcode field IMMEDIATE_DATA[11:8]). The internal register op_bytes_to_process_reg requires the option length in bytes (output of AU0). The internal register op_ca_enable_reg stores the state of the microcode field CA_ENABLE. This determines whether or not the option data is added to the checksum during option parsing. The LHB should not be popped when enabling the Option Parsing Unit.
The Option Parsing Unit "freezes" normal conditional branching by asserting the OP_NOT_DONE flag, which indicates that the Option Parsing Unit has not finished parsing all the options. Once the Option Parsing Unit deasserts the OP_NOT_DONE flag, the conditional branching control returns to the microcode.

Option Type Select	Option Type
0	IPv4
1	TCP

Table 8-34: Option Type Selection Values

8.15 Pseudo-header Register

The 128-bit Pseudo-header Register can stores data fields which need to be included in a checksum calculation. Each word of the Pseudo-header Register has 1 data input and four byte enables which allows each byte to be written independently. The data input can be selected (microcode field PHDR_WORDx_DATA_SELECT, see Table 8-35) from any of the four words of the LHB, or any of the four Holding Registers. The byte enables are controlled by the microcode fields LOAD_PHDR_WORDx. The output of the Pseudo-header Register can be fed as an input to the Checksum Accumulator.

Data Select Field	Data Field
0	LHB[127:96]
1	LHB[95:64]
2	LHB[63:32]
3	LHB[31:0]
4	{ 0x0000, Holding Register 0 }
5	{ 0x0000, Holding Register 1 }
6	{ 0x0000, Holding Register 2 }
7	{ 0x0000, Holding Register 3 }

Table 8-35: Pseudo-header Word Data Selection Values

8.16 Checksum Accumulator

The Checksum Accumulator calculates the ones compliment checksum from 128 bits. The Checksum Accumulator has a reset, an enable and 2 data sources. The reset (microcode field CA_RESET) clears the checksum and any internal registers. The enable allows a new checksum to be calculated from the data. If the Option Parsing Unit is enabled, the enable is controlled by the Option Parsing Unit. If the Option Parsing Unit is not enabled, the enable is controlled by the microcode field CA_ENABLE. The data can be selected (microcode field CA_DATA_SELECT, see Table 8-36) from the Pseudo-header Register or the LHB. Only the bytes that are to be popped from the LHB, as specified by the microcode field LHB_POP, are included in the checksum calculation. Therefore, any bits, on a byte basis, that are not to be included are cleared by the hardware. In the event that an odd number of bytes have been popped, the checksum accumulator adjusts accordingly. The checksum for the 128-bit input data is calculated and added to the existing checksum, producing a sixteen bit result called a "ca_result". This ca_result can then be stored in Event Register 1 as

the "running checksum". Also, the ca_result is compared to all ones and complimented, producing a single bit result called a "ca_result_invalid". This ca_result_invalid can then be stored in one of the sixteen Flag Registers, stored in one of the Event Flags, and/or used by the Abort Condition Gate.

Data Select Field	Data Field
0	LHB[127:0]
1	Pseudo-header Register

Table 8-36: Checksum Accumulator Data Selection Values

8.17 Priority Encoder

The Priority Encoder, shown below in Figure 8-10, is a 7-to-1 priority encoder that facilitates microcode branching. The Priority Encoder can be configured (microcode field PE_BRANCH_TYPE, see Table 8-37) to do a "case" branch, a "priority8" branch or a "priority4" branch. It has 8 inputs: condition0, condition1, condition2, condition3, condition4, condition5, condition6, ls_nxt_addr_bits. The condition0 input is the abort_result. The remaining condition inputs can be selected (microcode field PE_CONDITIONx_DATA_SELECT, see Table 8-38 through Table 8-43) from any of the compare_results, any of the cgate_results, any of the Flag Registers, any of the Abort Gate Register bits, the HSI code entry point bits, the logical processor number or op_not_done. The ls_nxt_addr_bits are the least significant three bits of the Control Store address extracted from the microcode field NEXT_ADDRESS. The Priority Encoder accomplishes microcode branching by replacing the least significant three bits of the Control Store address with three bits generated from the Priority Encoder based on its configuration and inputs.

If the Priority Encoder is configured to do a "case" branch, then the least significant three bits of the Control Store address are replaced with whatever bit combination exists on the condition4, condition5 and condition6 inputs. For example, if the condition4 and condition5 inputs are asserted and the condition6 input is deasserted, the least significant three bits would become "110".

If the Priority Encoder is configured to do a "priority8" branch, then the least significant three bits of the Control Store address are replaced with the 3-bit encoded value of the highest asserted condition input. For example, if the condition1, condition2 and condition4 inputs are all asserted, the least significant three bits would become "001". In the event that none of the condition inputs are asserted, the Priority Encoder restores the least significant three bits of the Control Store address present on the ls_nxt_addr_bits input.

If the Priority Encoder is configured to do a "priority4" branch, then the least significant two bits of the Control Store address are replaced with the 2-bit encoded value of the highest asserted condition input (inputs 0-2). For example, if the condition0 and condition2 are both asserted, the least significant two bits would become "00". In the event that none of the condition inputs are asserted, the Priority Encoder restores the least significant two bits of the Control Store address present on the ls_nxt_addr_bits input.

Branch Type Select	Branch Type
0	Priority 8
1	Priority 4
2	Case
3	Case

Table 8-37: Priority Encoder Branch Type Selection Values

PE Condition 1 Select	Condition Field	PE Condition 1 Select	Condition Field
0	Soft Compare Result 0	24	Flag Register 3
1	Soft Compare Result 1	25	Flag Register 4
2	Soft Compare Result 2	26	Flag Register 5
3	Soft Compare Result 3	27	Flag Register 6

4	Hard Compare Result 0	**28**	Flag Register 7
5	Hard Compare Result 1	**29**	Flag Register 8
6	Hard Compare Result 2	**30**	Flag Register 9
7	Hard Compare Result 3	**31**	Flag Register 10
8	Hard Compare Result 4	**32**	Flag Register 11
9	Hard Compare Result 5	**33**	Flag Register 12
10	Hard Compare Result 6	**34**	Flag Register 13
11	Hard Compare Result 7	**35**	Flag Register 14
12	Hard Compare Result 8	**36**	Flag Register 15
13	Hard Compare Result 9	**37**	Active Logical Processor
14	Hard Compare Result 10	**38**	Logic 0
15	AND Result 0	**39**	Abort Gate Register 0
16	AND Result 1	**40**	Abort Gate Register 1
17	AND Result 2	**41**	Abort Gate Register 2
18	AND Result 3	**42**	Abort Gate Register 3
19	OR Result 0	**43**	Option Parser ABORT
20	OR Result 1	**44-63**	Undefined
21	Flag Register 0		
22	Flag Register 1		
23	Flag Register 2		

Table 8-38: Priority Encoder Condition 1 Selection Values

PE Condition 2 Select	Condition Field	PE Condition 2 Select	Condition Field
0	Soft Compare Result 0	**24**	Flag Register 3
1	Soft Compare Result 1	**25**	Flag Register 4
2	Soft Compare Result 2	**26**	Flag Register 5
3	Soft Compare Result 3	**27**	Flag Register 6
4	Hard Compare Result 0	**28**	Flag Register 7
5	Hard Compare Result 1	**29**	Flag Register 8
6	Hard Compare Result 2	**30**	Flag Register 9
7	Hard Compare Result 3	**31**	Flag Register 10
8	Hard Compare Result 4	**32**	Flag Register 11
9	Hard Compare Result 5	**33**	Flag Register 12
10	Hard Compare Result 6	**34**	Flag Register 13
11	Hard Compare Result 7	**35**	Flag Register 14
12	Hard Compare Result 8	**36**	Flag Register 15
13	Hard Compare Result 9	**37**	Active Logical Processor
14	Hard Compare Result 10	**38**	Logic 0
15	AND Result 0	**39**	Abort Gate Register 0
16	AND Result 1	**40**	Abort Gate Register 1
17	AND Result 2	**41**	Abort Gate Register 2
18	AND Result 3	**42**	Abort Gate Register 3
19	OR Result 0	**43**	Option Parser NOT DONE
20	OR Result 1	**44-63**	Undefined
21	Flag Register 0		
22	Flag Register 1		
23	Flag Register 2		

Table 8-39: Priority Encoder Condition 2 Selection Values

PE		PE	

Condition 3 Select	Condition Field	Condition 3 Select	Condition Field
0	Soft Compare Result 0	24	Flag Register 3
1	Soft Compare Result 1	25	Flag Register 4
2	Soft Compare Result 2	26	Flag Register 5
3	Soft Compare Result 3	27	Flag Register 6
4	Hard Compare Result 0	28	Flag Register 7
5	Hard Compare Result 1	29	Flag Register 8
6	Hard Compare Result 2	30	Flag Register 9
7	Hard Compare Result 3	31	Flag Register 10
8	Hard Compare Result 4	32	Flag Register 11
9	Hard Compare Result 5	33	Flag Register 12
10	Hard Compare Result 6	34	Flag Register 13
11	Hard Compare Result 7	35	Flag Register 14
12	Hard Compare Result 8	36	Flag Register 15
13	Hard Compare Result 9	37	Active Logical Processor
14	Hard Compare Result 10	38	Logic 0
15	AND Result 0	39	Abort Gate Register 0
16	AND Result 1	40	Abort Gate Register 1
17	AND Result 2	41	Abort Gate Register 2
18	AND Result 3	42	Abort Gate Register 3
19	OR Result 0	43-63	Undefined
20	OR Result 1		
21	Flag Register 0		
22	Flag Register 1		
23	Flag Register 2		

Table 8-40: Priority Encoder Condition 3 Selection Values

PE Condition 4 Select	Condition Field	PE Condition 4 Select	Condition Field
0	Soft Compare Result 0	24	Flag Register 3
1	Soft Compare Result 1	25	Flag Register 4
2	Soft Compare Result 2	26	Flag Register 5
3	Soft Compare Result 3	27	Flag Register 6
4	Hard Compare Result 0	28	Flag Register 7
5	Hard Compare Result 1	29	Flag Register 8
6	Hard Compare Result 2	30	Flag Register 9
7	Hard Compare Result 3	31	Flag Register 10
8	Hard Compare Result 4	32	Flag Register 11
9	Hard Compare Result 5	33	Flag Register 12
10	Hard Compare Result 6	34	Flag Register 13
11	Hard Compare Result 7	35	Flag Register 14
12	Hard Compare Result 8	36	Flag Register 15
13	Hard Compare Result 9	37	Active Logical Processor
14	Hard Compare Result 10	38	Logic 0
15	AND Result 0	39	Abort Gate Register 0
16	AND Result 1	40	Abort Gate Register 1
17	AND Result 2	41	Abort Gate Register 2
18	AND Result 3	42	Abort Gate Register 3
19	OR Result 0	43	Abort Result
20	OR Result 1	44	Header Side Info 34
21	Flag Register 0	45	Microcode Next Address 2

22	Flag Register 1	**46-63**	Undefined
23	Flag Register 2		

Table 8-41: Priority Encoder Condition 4 Selection Values

PE Condition 5 Select	Condition Field	PE Condition 5 Select	Condition Field
0	Soft Compare Result 0	**24**	Flag Register 3
1	Soft Compare Result 1	**25**	Flag Register 4
2	Soft Compare Result 2	**26**	Flag Register 5
3	Soft Compare Result 3	**27**	Flag Register 6
4	Hard Compare Result 0	**28**	Flag Register 7
5	Hard Compare Result 1	**29**	Flag Register 8
6	Hard Compare Result 2	**30**	Flag Register 9
7	Hard Compare Result 3	**31**	Flag Register 10
8	Hard Compare Result 4	**32**	Flag Register 11
9	Hard Compare Result 5	**33**	Flag Register 12
10	Hard Compare Result 6	**34**	Flag Register 13
11	Hard Compare Result 7	**35**	Flag Register 14
12	Hard Compare Result 8	**36**	Flag Register 15
13	Hard Compare Result 9	**37**	Active Logical Processor
14	Hard Compare Result 10	**38**	Logic 0
15	AND Result 0	**39**	Abort Gate Register 0
16	AND Result 1	**40**	Abort Gate Register 1
17	AND Result 2	**41**	Abort Gate Register 2
18	AND Result 3	**42**	Abort Gate Register 3
19	OR Result 0	**43**	Abort Result
20	OR Result 1	**44**	Header Side Info 33
21	Flag Register 0	**45**	Microcode Next Address 1
22	Flag Register 1	**46-63**	Undefined
23	Flag Register 2		

Table 8-42: Priority Encoder Condition 5 Selection Values

PE Condition 6 Select	Condition Field	PE Condition 6 Select	Condition Field
0	Soft Compare Result 0	**24**	Flag Register 3
1	Soft Compare Result 1	**25**	Flag Register 4
2	Soft Compare Result 2	**26**	Flag Register 5
3	Soft Compare Result 3	**27**	Flag Register 6
4	Hard Compare Result 0	**28**	Flag Register 7
5	Hard Compare Result 1	**29**	Flag Register 8
6	Hard Compare Result 2	**30**	Flag Register 9
7	Hard Compare Result 3	**31**	Flag Register 10
8	Hard Compare Result 4	**32**	Flag Register 11
9	Hard Compare Result 5	**33**	Flag Register 12
10	Hard Compare Result 6	**34**	Flag Register 13
11	Hard Compare Result 7	**35**	Flag Register 14
12	Hard Compare Result 8	**36**	Flag Register 15
13	Hard Compare Result 9	**37**	Active Logical Processor
14	Hard Compare Result 10	**38**	Logic 0
15	AND Result 0	**39**	Abort Gate Register 0

16	AND Result 1	**40**	Abort Gate Register 1
17	AND Result 2	**41**	Abort Gate Register 2
18	AND Result 3	**42**	Abort Gate Register 3
19	OR Result 0	**43**	Abort Result
20	OR Result 1	**44**	Header Side Info 32
21	Flag Register 0	**45**	Microcode Next Address 0
22	Flag Register 1	**46-63**	Undefined
23	Flag Register 2		

Table 8-43: Priority Encoder Condition 6 Selection Values

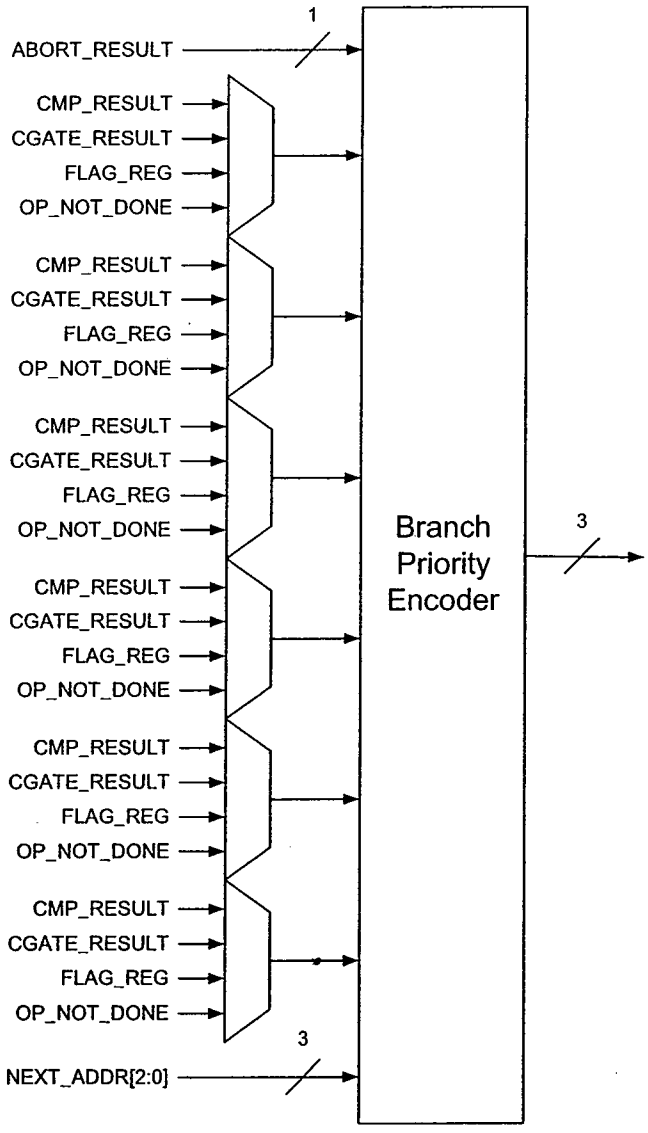

Figure 8-10: Priority Encoder

8.18 Control Store

The Control Store is a 428-bit by 192 entry RAM that holds all the microcode needed to process packets for a given Packet Processor configuration. The read address can be selected from two sources. If the byte_count_mismatch flag AND the microcode field LENGTH_ABORT_ENABLE are asserted, the output of the Length Abort Jump Address Register is selected. If the previous condition is not true, the concatenation of the microcode field NEXT_ADDRESS[6:3] and the 3-bit output of the Priority Encoder is selected.

8.19 Event Queue (EQ)

The Event Queue (EQ) resides outside the packet processor. The EQ is where the Packet Processor writes the Event Structure for the current packet being processed. The EQ is capable of buffering sixteen 16 quad-word Event Structures.

8.20 EQ Write Control

The EQ Write Control writes quad-words into the EQ as directed by the microcode. The EQ Write Control creates the 8-bit write address by concatenating the extracted 4 bits from the header side information (EVENT_NUM) and the selected 4-bit address field. The address field can be selected (microcode field EQ_ADDR_SELECT) from the least significant 4 bits of Holding Register 3 or the microcode field EQ_WR_ADDRESS. It also extracts the four word enables from the microcode field EQ_WRITE_ENABLE. The write enable bit is created by ORing the four word enable bits. The actual write data can be selected (microcode field EQ_DATA_SELECT) from one of the five Event Registers or one of eight halfword shifted combinations from the LHB. The exact selection values are listed below in Table 8-44 and Table 8-45 with one exception. Should the microcode fields LOAD_FRAME_ID and HOST_PACKET be asserted, bits 31-24 of the write data will be replaced with header side information (FRAME_ID).

EQ Address Select	EQ Write Address[3:0]
0	Microcode EQ Write Address Field
1	Holding Register 3[3:0]

Table 8-44: EQ Write Address Selection Values

EQ Data Select	EQ Write Data
0	Event Register 0
1	Event Register 1
2	Event Register 2
3	Event Register 3
4	Event Register 4
5	LHB[127:0]
6	{16'b0, LHB[127:16]}
7	{32'b0, LHB[127:32]}
8	{48'b0, LHB[127:48]}
9	{64'b0, LHB[127:64]}
10	{80'b0, LHB[127:80]}
11	{96'b0, LHB[127:96]}
12	{112'b0, LHB[127:112]}
13-15	Undefined

Table 8-45: EQ Write Data Selection Values

8.21 Miscellaneous

8.21.1 Logical Processor Number Register

The single bit Logical Processor Number Register indicates which logical processor is accessing the Control Store RAM. This register allows the logical processors to work independently and is used to control the sequential element source muxes and can also be used by the Priority Encoder.

8.21.2 Abort Condition Register

The 4-bit Abort Condition Register stores the 4 inputs to the Abort Condition Gate. These bits are updated every instruction cycle and are made available as inputs to the Priority Encoder. This allows for branching based on the abort condition that occurred.

8.21.3 Bytes Popped Register

The 8-bit Bytes Popped Register keeps track of how many bytes have been popped from the LHB. Each instruction cycle, the microcode field LHB_POP is added to the Bytes Popped Register. The output of the register is used by the HB Over-read Prevention Logic, as well as to load the Payload Scratch Offset and L3 Header Scratch Offset fields.

8.21.4 Bytes Remaining Register

The 16-bit Bytes Remaining Register keeps track of the number of bytes left to process. Each instruction cycle, the microcode field LHB_POP plus the Bytes Popped Register are subtracted from the total number of bytes to process. The total number of bytes to process is determined by the header side information field BYTE_CNT. If the byte count is zero, the total number of bytes to process is 256. If the byte count is not zero, the total number of bytes to process is the byte count. The output of the register is used by the arithmetic units as a data input.

8.21.5 HB Over-read Prevention Logic

The HB Over-read Prevention Logic ensures that the packet processor does not over-read the HB due to incorrect information extracted from the packet header. This is accomplished by adding the microcode field LHB_POP to the value contained in the Bytes Popped Register and comparing the result to the Header Side Info Register field BYTE_CNT, producing a single bit result called "byte_count_mismatch". If the sum of LHB_POP and the Bytes Popped Register is greater than BYTE_CNT, then byte_count_mismatch will be set. This byte_count_mismatch is then used by the Control Store. This is done on every instruction cycle and cannot be controlled by the microcode.

8.21.6 Header Side Info Register

The 45-bit Header Side Info (HSI) Register contains stores the Header Side Info that is presented at the start of each packet header. This register loads when HDR_AVAILABLE is asserted. The HSI Register is broken into sub-fields. The exact bit assignments for these sub-fields are shown below in Table 8-46.

Bits	Header Buffer I/F Signal Name
44	sp_data_val
43	force_sp_data
42:35	byte_cnt
34:32	code_entry_point
31:28	event_num
27:20	port_num
19:12	frame_id
11:0	pvid

Table 8-46: Header Side Info Register Bit Map

The various sub-fields are used differently by the packet processors. The sp_data_val and force_sp_data bits are used by the Event Registers logic to force a value into the SP Data bit. The byte_cnt field is used by the HB Over-read Prevention Logic. The code_entry_point field is used by the Priority Encoder to branch to the proper initial control word for the current packet header. The event_num field is used by the EQ Write Control logic to create the EQ write address. The port_num, frame_id and pvid fields are copied into the Event Registers, as required.

8.21.7 Freeze Logic

The Freeze Logic ensures that the packet processor does not begin processing a packet until a packet header is available and there is an entry available in the Event Queue. The hardware will "freeze" (execute the same microcode word) until both these conditions are true.

8.22 Estimated Cell Count

The estimated cell count, shown below in Table 8-47, is for 1 physical (2 logical) packet processor and is based on the current understanding of the packet processor hardware. As it is actually designed, this number will increase. A value of 3.17 was used as the gate-to-cell factor.

	Logic	Control Store (gates)	Control Store (cells)	Total Cells
Budget (entire system)	760,000	4KB	210,206	970,206
Estimate (1 physical)	420,544	192 X 428	452,996	873,540

Table 8-47: Estimated Cell Count

9 Verification Plan

The verification plan is structured as a list of features or conditions which need to be verified. This is followed by a description of the testbench which will be developed to enable the testing of those features, followed by a list of the tests which must be written to cover those features.

9.1 Features to be tested

1. Access of correct Header Buffer and LHB by each logical processor
2. Loading of Header Side Info Register
3. LHB popping of 0-16 bytes and corresponding re-filling
4. Verify Header Buffer is not over-read
5. Clearing PP registers and starting a new packet header
6. Selection of any contiguous 2 bytes from the LHB, any holding register or the bytes remaining value for a 16-bit soft compare
7. Selection of all ones or any Constant Mask Register for a 16-bit soft compare
8. Masking of each bit for a 16-bit soft compare
9. Selection of all zeros or any Constant Data Register as compare value for a 16-bit soft compare
10. Verify "==", "<" and ">" compares in each bit position for a 16-bit soft compare
11. Verify operation of the other 16-bit soft compares
12. Verify correct hard compares and selection (including flag registers)
13. Selection and inversion of any compare result (hard or soft) into any input of any AND gate
14. Selection and inversion of any compare result (hard or soft) into any input of any OR gate

15. Selection of defined subset into input and loading of all Flag Registers
16. Selection of defined subsets into inputs of the ABORT gate
17. Verify proper operation of Length Abort (i.e. Jump Address correctly used)
18. Selection of any contiguous 2 bytes from the LHB, any holding register, the bytes remaining value or the immediate value from the microcode for an arithmetic unit
19. Selection of all ones or any Constant Mask Register for an arithmetic unit
20. Masking of each bit for an arithmetic unit
21. Selection of any Constant Data Register or any holding register as a compare value for an arithmetic unit
22. Verify "+", "-" and "*4" operations for arithmetic unit 0
23. Verify "+", "-" and "/16" operations for arithmetic unit 1
24. Selection of any arithmetic unit result into any holding register
25. Verify Event Register Addressing
26. Selection of defined subsets into input and loading of all Event Flags
27. Forced setting of each Event Flag
28. Verify loading of all static fields for each header type in Event Registers
29. Proper Frame ID substitution for a host packet
30. Create Flow flag generation
31. Correct L3 Header Offset and Scratch Payload Offset calculation
32. Selection of Flow Key Receive I/F field
33. Selection of VLAN tag
34. Correct IP and TCP Option Length calculations
35. Verify Option Parsing Unit with various numbers and orders of options, including no options
36. Verify the Option Parsing Unit taking control from and restoring control to the microcode, with and without a 1-clock pause to wait for Logical Processor's correct clock cycle
37. Independent operation of each Option Parsing Unit, running individually
38. Independent operation of each Option Parsing Unit, both running simultaneously
39. Selection of LHB or any holding register for Pseudo-Header
40. Calculation of checksum
41. Selection of LHB (including masking) or Pseudo-Header for checksum accumulation
42. Clearing of checksum
43. Microcode branches: prioritized conditional with various flags on, including none and all, 8-way "case" conditional with various flags on
44. Proper addressing of Control Store
45. Writing of Event Queue with data from LHB or Event Registers
46. Writing of correct Event Queue by each logical processor
47. Popped Bytes accumulation / Bytes Remaining calculation
48. Stalling of each logical processor (due to lack of packet in Header Buffer or no space available in Event Queue) while other logical processor continues unaffected

49. Proper response to assertion of STOP bit in CONFIG register
50. Internal registers match spec definition
51. Proper incrementing and holding of general purpose counters for each logical processor
52. Proper Header Buffer Interface operation
53. Proper Event Queue Interface operation
54. Proper Initialization Interface operation
55. Proper operation when Event Queue space becomes available -2, -1, 0, +1, +2 clocks with respect to when needed
56. Proper operation when Header Buffer header becomes available -2, -1, 0, +1, +2 clocks with respect to when needed
57. Verify write/read operation to each Control Store Address and Word.

Table 9-1: Test Feature List

9.2 Testbench Organization and Features

The testbench will have the ability to respond to the packet processor's requests for packet header data (Header Buffer Interface), issue commands to write and read all registers and the control store RAM (Initialization Interface), and validate the generated Event Structure (Event Queue Interface). The testbench will be completely self-checking.

The Header Buffer contents, Control Store Memory image and expected Event Queue contents will be generated by other programs outside the verilog testbench, but written by Octera. These images will be used by the testbench to drive the Packet Processor and verify its results.

9.3 Test List

In general, these tests cause data to be written to the Event Queue, which is automatically verified by comparing the Event Queue to the expected data. In cases where a test produces results which cannot be directly written to the Event Queue, such as the output of a comparator or AND/OR gate, the microcode will branch on the resulting condition. Depending on where the microcode ends up, different data values are written to the Event Queue. This could be the Event Type, Event Subcode, Event Size or LHB data written to various Event Queue locations.

Note that the numbers in the "Features Covered" column are Word "Cross-reference"s to the features list in section 9.1.

Test Name	Description	Features covered
LHB_Pop	Verify that for any pop value (0-16), the new HB data can be aligned with any remaining data for any byte boundary. Verify that new HB data is requested only when 16+ bytes will become available. 17*17 = 289 cases	1, 3, 45, 46, 47, 53
New_Packet	Issue DONE command, check for clearing of LHB control logic, checksum, pseudo-header, and any other registers which should be cleared. Check for PP_DONE. Verify that when header is available, header side info register is loaded. 1 case	1, 2, 5
SC_Select	Verify that all possible LHB halfwords (16), all possible holding registers (4) and the bytes remaining value can be selected for the data input of the 16-bit soft compares. Verify that all possible Constant Data Registers (17) plus all zeros can be selected for the constant input of the 16-bit soft compares. 21 cases	6, 9
SC_Mask	Verify that all Constant Mask Registers (7) plus all ones can be selected for the mask input of the 16-bit soft compares. Verify that each data bit can be masked off individually and as a group. 26 cases	7, 8

SC_Operation	Verify that each operation type can be selected and that it functions properly (true & false). 30 cases (5 sets, compare true/false, "==", "<", ">").	10, 11
HC_All	Verify that all hard comparators function. Verify selection of each group. 14 cases (7 groups for a total of 47 compares, compare true/false)	12
AU_Select	Verify that all possible LHB halfwords (16), all possible holding registers (4), the bytes remaining value and the microcode immediate value can be selected for the data input of the 16-bit arithmetic units. Verify that all possible Constant Data Registers (12) and all possible holding registers (4) can be selected for the constant input of the 16-bit arithmetic units. 21 cases	18, 21
AU_Mask	Verify that all Constant Mask Registers (7) plus all ones can be selected for the mask input of the 16-bit arithmetic units. Verify that each data bit can be masked off individually and as a group. 26 cases	19, 20
AU_Operation	Verify that each operation type can be selected and that it functions properly. 24 cases (4 sets, "+", "-", "*4", "/16").	22, 23
AND_OR	Verify that all possible compare results (15) can be selected as an input to any AND or OR gate (24 inputs). Also verify inversion and disabling of (unused) inputs. 120 cases	13, 14
Abort	Verify that all possible subset elements (8) can be selected as an input to the Abort OR gate (4 inputs). Also verify disabling of (unused) inputs. Verify the proper operation of the length abort and the corresponding jump address. 33 cases	16, 17
Flag_Register	Verify that all possible subset elements (4) can be selected and loaded into the corresponding flag register (16). 10 cases	15
Holding_Register	Verify that all possible arithmetic unit results can be selected and loaded into any holding register (4). 1 case	24
Event_Register	Verify that all static fields in the Event Register (flag field, create flow, event code, encoded protocol, size, subcode, scratch (data) offset, SP/DP, SA, DA, VLAN ID, switch fabric header length, IP and TCP option lengths and offsets, checksum) can be generated and loaded. ~30-50 cases	25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 45
PH_Register	Verify that all possible LHB words can be selected and loaded (on a byte basis) into the Pseudo-Header Register. 24 cases	39
CheckSum	Verify that all possible data sources (2) can be selected to calculate a new checksum. Verify that the checksum can be cleared. 6 cases	40, 41, 42
Option_Parse	Check Option Parsing Unit with various numbers and orders of options, including no options. Verify that each Option Parsing Unit can operate independently of the other, including taking control from and restoring control to the microcode. This will also include various edge conditions such as: OP1 taking control one clock after OP0 has taken control OP0 taking control one clock after OP1 has taken control OP1 restoring control one clock after OP0 has restored control OP0 restoring control one clock after OP1 has restored control OP0 and OP1 restoring control on the same clock. OP0/1 completes 1 266MHz clock before its logical processor is available, and simultaneously 18 cases	35, 36, 37, 38
Priority_Branch	Verify that all possible input sources (45) can be selected for each input (6). Verify that each branch type (2) can be selected and that branching occurs as selected. Verify that each priority input is unaffected by lower priority inputs. Verify priority fall-out case (no inputs asserted). Verify each bit combination for case branch. 407 cases	43, 44
HB_Overreads	Verify that over-reading the Header Buffer based on packet header info for a frame that is completely in the Header Buffer causes an abort. 5 cases	4, 52
Freeze	Verify that each logical processor will properly "freeze" when space is not available in the Event Queue and/or no packet header is available. Verify that each can run independently of the other and that each properly responds when its STOP bit is asserted or deasserted in the CONFIG register. This will include various edge conditions such as: STOP asserted/deasserted –2, -1, 0, +1, +2 clocks with respect to DONE STOP asserted/deasserted –2, -1, 0, +1, +2 clocks with respect to HDR_AVAIL STOP asserted/deasserted –2, -1, 0, +1, +2 clocks with respect to EQ_RDY	48, 49, 55, 56

	HDR_AVAIL asserted –2, -1, 0, +1, +2 clocks with respect to when needed EQ_RDY asserted –2, -1, 0, +1, +2 clocks with respect to when needed 40 cases	
Internal_Registers	Verify that all internal registers are accessible via initialization interface as defined. 18 cases	50, 54, 57
Counters	Verify that all counters can be incremented and don't rollover. 1 case	51, 54
Astute_Test	Verify that 1 Astute generated test (up to 5 packet headers & event structures) and the processing algorithm runs as expected. 1 case	

Table 9-2: Test List

10 Open Issues

None.

11 Summary

The preceding sections should have accurately described the operation of the packet processor. Please notify the author of discrepancies, omissions or typos.

Appendix A

The following pseudo-code may be helpful in understanding the capabilities of the VLIW Packet Processor hardware. It is also the method used to estimate the Packet Processor's performance.

This pseudo-code was written before the architecture was created. It defines the kinds of operations required of the hardware, and was a driving factor in creating the architecture. It is not intended to be a complete or correct representation of the final Microcode.

Syntax Legend:

```
X                   X is a specific field in the LHB
X -> Y              Y takes on the value of X
(X) ? ...           If X is true then ...
X_reg               Internal register X
ES.X.Y              X event structure, field Y
LHB << Y            Y bytes are popped from the LHB
set X flag          X is a temporary flag that during current cycle only
hreg.X              X (convenient name) is in a holding register
freg.X              X (convenient name) is in a flag register
NEXT_ADDR(X,"Y")    Conditional branch in which X is highest priority and Y is the
                      fall out case (no priority inputs asserted)
```

When "hdr_available" transitions from 0 -> 1, begin header processing...

Startup

```
byte_cnt -> frame_length_reg,
(!full_frame) ? set frame_frag_reg,
frame_id -> ES.control.frame_id,
event_num -> eq_wr_addr_ms_bits_reg,
port_num -> ES.mac.spi4_port_num,
frame_err -> ES.mac_flags.frame_error,
pvid -> pvid_reg,
NEXT_ADDR(code_entry_point) -> CS RAM Addr:
```

Host Header0 (CF = b0[7], ET = b0[5:0], FK = b3[3:0])

```
CF -> ES.control.cf,
ET -> ES.control.event_type,
FK -> ES.control.fk_rcv_if[11:8],
byte(4) thru byte(15) -> Event Queue(MC_eq_wr_addr),
LHB << 16,
NEXT_ADDR -> CS RAM Addr:
```

Host Header1 (FK = b0-b1, ESZ = b2[4:0], ESC = b3, SID = b5[5:0]-b7)

```
FK -> {ES.control.fk_rcv_if[7:0], ES.control.fk_protocol},
(ESZ + 2) -> ES.control.event_size,
ESC -> ES.control.event_subcode,
(ESZ - 1) -> hreg.data_length,
(ESZ == 0) ? set NO_DATA flag,
byte(4) thru byte(7) -> Event Queue(MC_eq_wr_addr),
LHB << 8,
NEXT_ADDR(NO_DATA) -> CS RAM Addr:
```

Host Data (DATA = b0-b15)

```
(hreg.data_length == 0) ? set NO_DATA flag,
(hreg.data_length - 1) -> hreg.data_length,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
```

```
LHB << 16,
NEXT_ADDR(NO_DATA) -> CS RAM Addr:
```

Event Register (Control0)

```
event_reg(0) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control1)

```
1 -> pp_done,
1 -> ip_frm_end,
event_reg(1) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Switch Fabric

```
*byte(0) thru byte(MC_lhb_pop) -> Event Queue(MC_eq_wr_addr),
*MC_lhb_pop -> bytes_popped_reg, ES.mac.fabhdr_len,
*LHB << MC_lhb_pop,
*NEXT_ADDR -> CS RAM Addr:
```

MAC address and type check (DA = b0-b5, SA = b6-b11, TYPE = b12-b13, DSAP_SSAP = b14-b15)

```
(DA == <-1>) ? set BCAST flag, set freg.bcast, set ES.mac.mac_bcast_i,
(DA[40] == 1) ? set MCAST flag, set freg.mcast, set ES.mac.mac_mcast_i,
(DA & PMAC_MASK) == EXP_PMAC_DA) ? set freg.pmac,
(((DA & TMAC_MASK) != EXP_TMAC_DA) && !BCAST && !MCAST) ? set CHECK_PMAC flag,
                                                          set ES.mac.tmac_addr_e,
(TYPE == 0x8100) ? set VLAN flag, set ES.mac.vlan_i,
(TYPE == 0x0800) ? set IPv4 flag,
(TYPE == 0x0806) ? set ARP flag,
(TYPE  > 0x05DC) ? set ETHER flag, set ES.mac.ether_i,
(ETHER && !IPv4 && !ARP) ? set ES.mac.mac_prot_e,
(!ETHER && (DSAP_SSAP == 0xAAAA)) ? set SNAP flag,
(!ETHER && (DSAP_SSAP != 0xAAAA)) ? set SNAP_ERR flag, set ES.mac.snap_e,
byte(12) thru byte(13) -> ES.control.protocol,
byte(0) thru byte(11) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg + 14 -> bytes_popped_reg,
LHB << 14,
NEXT_ADDR(CHECK_PMAC,VLAN,SNAP,SNAP_ERR,IPv4,ARP,"UNREC PROTOCOL") -> CS RAM Addr:
```

PMAC address check

```
(!freg.pmac) ? set ABORT flag, set ES.mac.pmac_addr_e,
LHB << 0,
NEXT_ADDR(ABORT,"UNREC PROTOCOL") -> CS RAM Addr:
```

VLAN present (PRI = b0[7:5], CFI = b0[4], VLAN ID = b0[3:0]-b1, TYPE = b2-b3, DSAP_SSAP = b4-b5)

```
(CFI == 1) ? set CFI flag,
(TYPE == 0x0800) ? set IPv4 flag,
(TYPE == 0x0806) ? set ARP flag,
(TYPE  > 0x05DC) ? set ETHER flag, set ES.mac.ether_i,
(ETHER && !IPv4 && !ARP) ? set ES.mac.mac_prot_e,
(!ETHER && (DSAP_SSAP == 0xAAAA)) ? set SNAP flag,
(!ETHER && (DSAP_SSAP != 0xAAAA)) ? set ABORT flag, set ES.mac.snap_e,
byte(0) thru byte(1) -> {ES.mac.priority, ES.mac.cfi, ES.mac.vlan_id},
byte(2) thru byte(3) -> ES.control.protocol,
bytes_popped_reg + 4 -> bytes_popped_reg,
LHB << 4,
NEXT_ADDR(ABORT,CFI,ETHER,SNAP,IPv4,ARP,"UNREC PROTOCOL") -> CS RAM Addr:
```

SNAP present (DSAP_SSAP = b0-b1, UI_OUI = b2-b5, TYPE = b6-b7)

```
(UI_OUI != 0x03000000) ? set ABORT flag, set ES.mac.ui_oui_e,
(TYPE == 0x0800) ? set IPv4 flag,
(TYPE == 0x0806) ? set ARP flag,
(!IPv4 && !ARP) ? set_ES.mac.mac_prot_e,
*pvid_reg -> ES.mac.vlan_id,
byte(6) thru byte(7) -> ES.control.protocol,
byte(2) thru byte(5) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg + 8 -> bytes_popped_reg,
LHB << 8,
NEXT_ADDR(ABORT,IPv4,ARP,"UNREC PROTOCOL") -> CS RAM Addr:
```

Unrecognized L3 Protocol (DATA = b0-b15)

```
HOW KNOW WHEN TO STOP?
(frame_frag_reg && (bytes_popped_reg < frame_length_reg)) ? set MORE_DATA flag,
"proto_unrec" -> EF.control.event_type,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
*bytes_popped_reg -> ES.mac.l3_header_offset,
bytes_popped_reg + 16 -> bytes_popped_reg,
LHB << 16,
NEXT_ADDR(MORE_DATA) -> CS RAM Addr:
```

Event Register (MAC0)

```
(MODE A) ? 0 -> ES.control.rcv_if,
(MODE B) ? ES.mac.vlan_id[11:0] -> ES.control.rcv_if,
(MODE C) ? ES.mac.spi4_port_num -> ES.control.rcv_if,
event_reg(2) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC1)

```
event_reg(3) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control0)

```
event_reg(0) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control1)

```
1 -> pp_done,
1 -> ip_frm_end,
event_reg(1) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

ARP frame start (HW_TYPE = b0-b1, P_TYPE = b2-b3, L2A_LEN = b4, L3A_LEN = b5, OPERATION = b6-b7, SND_L2_ADR = b8-b13)

```
*(freg.bcast || freg.mcast)  ? set ABORT flag,
(P_TYPE != EXP_PROTOCOL) ? set P_TYPE_ERR flag,
(L2A_LEN != EXP_L2A_LEN) ? set L2A_LEN_ERR flag,
(L3A_LEN != EXP_L3A_LEN) ? set L3A_LEN_ERR flag,
(OPERATION != 0x01) && (OPERATION != 0x02)) ? set ABORT flag, set ES.arp.arp_op_e,
(P_TYPE_ERR || L2A_LEN_ERR || L3A_LEN_ERR) ? set ABORT flag, set ES.arp.arp_prot_e,
bytes_popped_reg -> ES.mac.l3_header_offset,
"arp" -> ES.control.event_type,
byte(0) thru byte(11) -> Event Queue(MC_eq_wr_addr),
LHB << 12,
NEXT_ADDR(ABORT) -> CS RAM Addr:
```

ARP frame end (SND_L2_ADR[15:0] = b0-b1, SND_L3_ADR = b2-b5, TGT_L2_ADR = b6-b11, TGT_L3_ADR = b12-b15)

```
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
LHB << 16,
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC0)

```
(MODE A) ? 0 -> ES.control.fk_rcv_if,
(MODE B) ? ES.mac.vlan_id[11:0] -> ES.control.fk_rcv_if,
(MODE C) ? ES.mac.spi4_port_num -> ES.control.fk_rcv_if,
event_reg(2) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC1)

```
event_reg(3) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (ARP)

```
event_reg(4) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control0)

```
event_reg(0) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control1)

```
1 -> pp_done,
1 -> ip_frm_end,
event_reg(1) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

IP frame description (VERSION = b0[7:4], HDR_LEN = b0[3:0], TOS = b1, TOT_LEN = b2-b3, ID = b4-b5, DF = b6[6], MF = b6[5], FOFF = b6[4:0]-b7, TTL = b8, PTCL = b9, HDR_CS = b10-b11, SA = b12-b15)

```
*(freg.bcast || freg.mcast) ? set ABORT flag,
(VERSION != 0x4) ? set ABORT flag, set ES.ipv4.ip_ver_e,
(HDR_LEN < 0x5) ? set ABORT flag, set ES.ipv4.ip_short_e,
(HDR_LEN > 0x5) ? set OPTIONS flag,
(TOT_LEN < (HDR_LEN << 2)) ? set ABORT flag, set ES.ipv4.ip_tot_len_e,
(TOT_LEN - (HDR_LEN << 2)) -> ip_data_length_reg,
(HDR_LEN - 5) -> hreg.option_length, ES.ipv4.opt_len,
((MF == 1) && (FOFF == 0)) ? set ES.ipv4.frag_f,
((MF == 0) && (FOFF != 0)) ? set ES.ipv4.frag_l,
((MF == 1) || (FOFF != 0)) ? set ES.ipv4.frag_i,
                             set freg.no_cs,
                             "ip_fragment" -> ES.control.event_type,
(PTCL == 0x01) ? set ICMP flag, set freg.icmp,
(PTCL == 0x06) ? set TCP flag, set freg.tcp,
(PTCL == 0x11) ? set UDP flag, set freg.udp,
(!ICMP && !TCP && !UDP) ? set freg.unknown, set ES.ipv4.ip_prot_e,
((FOFF != 0) || ICMP) ? set freg.no_phdr,
(TCP && OPTIONS) ? set OPTIONS_AND_TCP flag,
byte(0) thru byte(3) -> Checksum Accumulator,
byte(0) thru byte(3) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg -> ES.mac.l3_header_offset,
bytes_popped_reg + 4 -> bytes_popped_reg,
LHB << 4,
NEXT_ADDR(ABORT,OPTIONS_AND_TCP,OPTIONS) -> CS RAM Addr:
```

IP addresses and NO options (ID = b0-b1, DF = b2[6], MF = b2[5], FOFF = b2[4:0]-b3, TTL = b4, PTCL = b5, HDR_CS = b6-b7, SA = b8-b11, DA = b12-b15)

```
hreg.ip_data_length -> phdr_reg(word1),
byte(5) -> phdr_reg(word0), ES.control.fk_protocol,
byte(8) thru byte(11) -> phdr_reg(word2), ES.control.fk_sa,
byte(12) thru byte(15) -> phdr_reg(word3), ES.control.fk_da,
byte(0) thru byte(15) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg + 16 -> bytes_popped_reg,
LHB << 16,
NEXT_ADDR(freg.no_phdr) -> CS RAM Addr:
```

IP addresses and options (ID = b0-b1, DF = b2[6], MF = b2[5], FOFF = b2[4:0]-b3, TTL = b4, PTCL = b5, HDR_CS = b6-b7, SA = b8-b11, DA = b12-b15)

```
(hreg.option_length << 2) -> opt_par_opt_bytes_to_process_reg,
hreg.ip_data_length -> phdr_reg(word1),
byte(5) -> phdr_reg(word0), ES.control.fk_protocol,
byte(8) thru byte(11) -> phdr_reg(word2), ES.control.fk_sa,
byte(12) thru byte(15) -> phdr_reg(word3), ES.control.fk_da,
byte(0) thru byte(15) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg + 16 -> bytes_popped_reg,
LHB << 16,
NEXT_ADDR -> CS RAM Addr:
```

IP options (option parsing logic enabled with option select = IP) (OPT = b0-b15)

```
byte(0) thru byte(OPT_PAR_LHB_POP-1) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(OPT_PAR_EQ_WR_ADDR),
bytes_popped_reg + OPT_PAR_LHB_POP -> bytes_popped_reg,
LHB << OPT_PAR_LHB_POP,
NEXT_ADDR(OPT_PAR_NOT_DONE,freg.no_phdr) -> CS RAM Addr:
```

IP checksum and NO pseudo-header

```
(checksum_reg != 0xFFFF) ? set ABORT flag, set ES.ipv4.ip_csum_e,
NEXT_ADDR(ABORT,freg.tcp,freg.udp,freg.icmp,freg.unknown) -> CS RAM Addr:
```

IP checksum and pseudo-header

```
(checksum_reg != 0xFFFF) ? set ABORT flag, set ES.ipv4.ip_csum_e,
NEXT_ADDR(ABORT,freg.tcp,freg.udp,freg.icmp,freg.unknown) -> CS RAM Addr:
```

Unrecognized L4 Protocol (DATA = b0-b15)

```
HOW KNOW WHEN TO STOP?
(frame_frag_reg && (bytes_popped_reg < frame_length_reg)) ? set MORE_DATA flag,
"ip_unrec" -> ES.control.event_code,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
*bytes_popped_reg -> ES.control.payload_scratch_offset,
bytes_popped_reg + 16 -> bytes_popped_reg,
LHB << 16,
NEXT_ADDR(MORE_DATA) -> CS RAM Addr:
```

Event Register (MAC0)

```
(MODE A) ? 0 -> ES.control.rcv_if,
(MODE B) ? ES.mac.vlan_id[11:0] -> ES.control.fk_rcv_if,
(MODE C) ? ES.mac.spi4_port_num -> ES.control.fk_rcv_if,
event_reg(2) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC1)

```
event_reg(3) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (IPv4)

```
event_reg(5) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control0)

```
event_reg(0) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control1)

```
1 -> pp_done,
1 -> ip_frm_end,
event_reg(1) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

TCP pseudo-header (SP = b0-b1, DP = b2-b3, SEQ_NUM = b4-b7, ACK = b8-b11, HDR_LEN = b12[7:4], FLAGS = b13[5:0], WIN = b14-b15)

```
(freg.bcast || freg.mcast) ? set ABORT flag, set ES.tcp.tcp_bcast_e,
(hreg.ip_data_length < 0x0014) ? set ABORT flag, set ES.tcp.tcp_ip_len_e,
(hreg.ip_data_length - (HDR_LEN << 2)) -> hreg.data_length,
*phdr_reg -> Checksum Accumulator,
NEXT_ADDR(ABORT) -> CS RAM Addr:
```

TCP description (SP = b0-b1, DP = b2-b3, SEQ_NUM = b4-b7, ACK = b8-b11, HDR_LEN = b12[7:4], FLAGS = b13[5:0], WIN = b14-b15)

```
(HDR_LEN < 0x5) ? set ABORT flag, set ES.tcp.tcp_short_e,
(hreg.data_length == 0x0000) ? set freg.no_data, set ES.tcp.tcp_nodata_i,
((FLAGS && CF_MASK) == EXP_FLAGS) ? set ES.control.cf,
(HDR_LEN - 5) -> hreg.tcp_option_length, ES.tcp.opt_len,
"tcp" -> ES.control.event_type,
byte(0) thru byte(1) -> ES.control.fk_sp,
byte(2) thru byte(3) -> ES.control.fk_dp,
byte(0) thru byte(15) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg + 16 -> bytes_popped_reg,
LHB << 16,
NEXT_ADDR(ABORT) -> CS RAM Addr:
```

TCP checksum, upointer and NO options (CS = b0-b1, UP = b2-b3)

```
byte(0) thru byte(3) -> Checksum Accumulator,
byte(0) thru byte(3) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg + 4 -> bytes_popped_reg,
LHB << 4,
NEXT_ADDR(no_data_reg) -> CS RAM Addr:
```

TCP checksum, upointer and options (CS = b0-b1, UP = b2-b3)

```
(hreg.tcp_option_length << 2) -> opt_par_opt_bytes_to_process_reg,
byte(0) thru byte(3) -> Checksum Accumulator,
byte(0) thru byte(3) -> Event Queue(MC_eq_wr_addr),
bytes_popped_reg + 4 -> bytes_popped_reg,
LHB << 4,
NEXT_ADDR -> CS RAM Addr:
```

TCP options (option parsing logic enabled with option select = TCP) (OPT = b0-b15)

```
(hreg.data_length > 0x0010) ? set DATA_GT_16 flag,
byte(0) thru byte(OPT_PAR_LHB_POP-1) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(OPT_PAR_EQ_WR_ADDR),
bytes_popped_reg + OPT_PAR_LHB_POP -> bytes_popped_reg,
LHB << OPT_PAR_LHB_POP,
NEXT_ADDR(OPT_PAR_NOT_DONE,freg.no_data,DATA_GT_16) -> CS RAM Addr:
```

TCP data greater than 16 (DATA = b0-b15)

```
(hreg.data_length > 0x0010) ? set DATA_GT_16 flag,
byte(0) thru byte(15) -> Checksum Accumulator,
(hreg.data_length - 16) -> hreg.data_length,
*bytes_popped_reg -> ES.control.payload_scratch_offset,
LHB << 16,
NEXT_ADDR(DATA_GT_16) -> CS RAM Addr:
```

TCP data NOT greater than 16 (DATA = b0-b15)

```
byte(0) thru byte(15) -> Checksum Accumulator,
LHB << 16,
NEXT_ADDR -> CS RAM Addr:
```

*** Continued on next page ***

Event Register (MAC0)

```
(MODE A) ? 0 -> ES.control.fk_rcv_if,
(MODE B) ? ES.mac.vlan_id[11:0] -> ES.control.fk_rcv_if,
(MODE C) ? ES.mac.spi4_port_num -> ES.control.fk_rcv_if,
event_reg(2) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC1)

```
event_reg(3) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (IPv4)

```
event_reg(5) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (TCP)

```
event_reg(8) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control0)

```
event_reg(0) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control1)

```
1 -> pp_done,
1 -> ip_frm_end,
event_reg(1) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

UDP pseudo-header (SP = b0-b1, DP = b2-b3, TOT_LEN = b4-b5, CS = b6-b7, DATA = b8-b11)

```
*(freg.bcast || freg.mcast) ? set ABORT flag,
(hreg.ip_data_length < 0x0008) ? set ABORT flag,
(hreg.ip_data_length - 8) -> hreg.data_length,
*phdr_reg -> Checksum Accumulator,
NEXT_ADDR(ABORT) -> CS RAM Addr:
```

UDP description (SP = b0-b1, DP = b2-b3, TOT_LEN = b4-b5, CS = b6-b7, DATA = b8-b11)

```
(TOT_LEN > hreg.ip_data_length) ? set ABORT flag, set ES.udp.udp_len_e,
(TOT_LEN < 0x0008) ? set ABORT flag, set ES.udp.udp_short_e,
(hreg.data_length - 4) -> hreg.data_length,
(hreg.data_length  < 0x0005) ? set NO_DATA flag,
((hreg.data_length - 4)  > 0x0010) ? set DATA_GT_16 flag,
(CS == 0x0000) ? set freg.no_cs,
"udp" -> ES.control.event_type,
byte(0) thru byte(1) -> ES.control.fk_sp,
byte(2) thru byte(3) -> ES.control.fk_dp,
byte(0) thru byte(11) -> Checksum Accumulator,
byte(0) thru byte(11) -> Event Queue(MC_eq_wr_addr),
LHB << 12,
NEXT_ADDR(ABORT, NO_DATA, DATA_GT_16) -> CS RAM Addr:
```

UDP data greater than 16 (DATA = b0-b15)

```
(hreg.data_length > 0x0010) ? set DATA_GT_16 flag,
byte(0) thru byte(15) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
(hreg.data_length - 16) -> hreg.data_length,
*bytes_popped_reg -> ES.control.payload_scratch_offset,
LHB << 16,
NEXT_ADDR(DATA_GT_16) -> CS RAM Addr:
```

UDP data NOT greater than 16 (DATA = b0-b15)

```
byte(0) thru byte(15) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
*bytes_popped_reg -> ES.control.payload_scratch_offset,
LHB << 16,
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC0)

```
(MODE A) ? 0 -> ES.control.fk_rcv_if,
(MODE B) ? ES.mac.vlan_id[11:0] -> ES.control.fk_rcv_if,
(MODE C) ? ES.mac.spi4_port_num -> ES.control.fk_rcv_if,
event_reg(2) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC1)

```
event_reg(3) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (IPv4)

```
event_reg(5) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (UDP)

```
event_reg(7) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control0)

```
event_reg(0) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control1)

```
1 -> pp_done,
1 -> ip_frm_end,
event_reg(1) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

ICMP description (TYPE = b0, CODE = b1, CS = b2-b3, DATA = b4-b11)

```
*(freg.bcast || freg.mcast) ? set ABORT flag,
(hreg.ip_data_length < 0x0004) ? set ABORT flag, set ES.icmp.icmp_short_e,
(hreg.ip_data_length - 12) -> hreg.data_length,
((hreg.ip_data_length - 4) < 0x0009) ? set NO_DATA flag,
((hreg.ip_data_length - 12) > 0x0010) ? set DATA_GT_16 flag,
"icmp" -> ES.control.event_type,
byte(0) thru byte(11) -> Checksum Accumulator,
byte(0) thru byte(11) -> Event Queue(MC_eq_wr_addr),
LHB << 12,
NEXT_ADDR(ABORT, NO_DATA, DATA_GT_16) -> CS RAM Addr:
```

ICMP data greater than 16 (DATA = b0-b15)

```
(hreg.data_length > 0x0010) ? set DATA_GT_16 flag,
byte(0) thru byte(15) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
(hreg.data_length - 16) -> hreg.data_length,
*bytes_popped_reg -> ES.control.payload_scratch_offset,
LHB << 16,
NEXT_ADDR(DATA_GT_16) -> CS RAM Addr:
```

ICMP data NOT greater than 16 (DATA = b0-b15)

```
byte(0) thru byte(15) -> Checksum Accumulator,
byte(0) thru byte(15) -> Event Queue(MC_eq_wr_addr),
*bytes_popped_reg -> ES.control.payload_scratch_offset,
LHB << 16,
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC0)

```
(MODE A) ? 0 -> ES.control.fk_rcv_if,
(MODE B) ? ES.mac.vlan_id[11:0] -> ES.control.fk_rcv_if,
(MODE C) ? ES.mac.spi4_port_num -> ES.control.fk_rcv_if,
event_reg(2) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (MAC1)

```
event_reg(3) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (IPv4)

```
event_reg(5) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (ICMP)

```
event_reg(6) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control0)

```
event_reg(0) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Event Register (Control1)

```
1 -> pp_done,
1 -> ip_frm_end,
event_reg(1) -> Event Queue(MC_eq_wr_addr),
NEXT_ADDR -> CS RAM Addr:
```

Appendix B

The op-code syntax described here is an example of how the Packet Processor control word could be created.

Notation:

Normal_Text	Syntactical element
Bold	Literal words and characters
Italics	Fields with a range of permissable values
|	
[...]	Restrictions, such as range of values permitted in *italic* field, or short description
(*i*)	Index into an array of possible fields
(... | ...)	List of possible options, one of which must be used
<...>	Descriptive name of field
{...}	Optional element

Syntax:

Program: OP_Code | Code_List
Code_List: (OP_Code | Compiler_Directive | Comment) {Code_List}
OP_Code:

Label:	
Done,	
Drop_Frame,	
Soft_Compare(*i*) := (LHB_Start_Byte(*j*) | Holding_Register(*k*))	[*i* = 0-3, *j* = 0-14, *k* = 0-3]
{&& Constant_Mask(*l*)} (== | < | >) Constant_Data(*m*),	[*l* = 0-6, *m* = 0-14]
AU(*ii*) := (**LHB_Start_Byte(*jj*) | Holding_Register(*kk*)** | <16-bit field>)	[*ii* = 0-1, *jj* = 0-14, *kk* = 0-3]
{ && **Constant_Mask(*ll*)** } { *4 | ((+ | -) (**Constant_Data(*mm*) | Holding_Register(*nn*)**)) },	[*ll* = 0-6] [*mm* = 0-14, *nn* = 0-3]
Holding_Register(*pp*) := AU(*qq*),	[*pp* = 0-3, *qq* = 0-1]
AND(*m*) := gate_list,	[*m* = 0-3]
OR(*n*) := gate_list,	[*n* = 0-1]
ER_Flag_Register := *rr*,	[*rr* = 0-5]
Flag_Register(*p*) := (**AND(*q*)** | Compare_Result | OR(r) | CA_RSLT_VALID),	[*p* = 0-15, *q* = 0-3] [*r* = 0-1]
Event_Flag(*p*) := (**AND(*q*)** | Compare_Result), OR(r) | CA_RSLT_VALID),	[*p* = 0-15, *q* = 0-3] [*r* = 0-1]
Load_Create_Flow_Flag,	
Host_Packet,	
Event_Type := <6-bit field>,	
Event_Size := <5-bit field>,	
Load_Flow_Key_ReceiveIF_Upper,	
Load_Flow_Key_ReceiveIF_Lower,	
Load_Flow_Key_Protocol,	
Load_Flow_Key_SP_DP,	
Load_Flow_Key_SA,	
Load_Flow_Key_DA,	
Load_Event_Frame_ID,	
Load_Event_IPU_SPI4_Port_Nums,	
Load_Event_L3_Header_Offset,	
Load_Event_Checksum,	
Event_Subcode := <8-bit field>,	
Option_Offset := <8-bit field>,	

Load_Event_Scratch_Payload_Offset,
VLAN_Tag := (LHB | PVID), [LHB(0-1) assumed]
Load_Switch_Fabric_Header_Length, [loads LHB_Pop *constant*]
Inc_Counter(*cc*), [*cc* = 1-7]
Abort := abort_list,
LHB_Pop *constant*, [*constant* = 0-16]
Pseudo_Header_Word(*u*) := { <4-bit field> **&&** } [*u* = 0-3]
(**LHB_Word(*v*) | Holding_Register(*w*)**), [*v* = 0-3, *w* = 0-3]
CheckSum_Accumulate := **LHB | Pseudo_Header,**
CheckSum_Clear,
IP_Option_Length = **AU(*x*),** [*x* = 0-1]
TCP_Option_Length = AU(*y*), [*y* = 0-1]
Option_Parse (IP | TCP), [starts option-parsing]
Event_Queue(*aa*) := { <4-bit word-mask> **&&** } [*aa* = 0-15]
(**LHB_Word_Shift(*bb*) | Event_Register(*cc*)**), [*bb* = 0-3, *cc* = 0-5]
Next_Address := Label,
Branch_Condition(*qq*) := (**AND(*rr*) | OR(*ss*) | Flag_Reg(*tt*)** [*qq* = 0-5, *rr* = 0-3, *ss* = 0-1,
| Compare_Result | **Start_Address_Offset),** *tt* = 0-15]
Branch_Case;

Compiler_Directive: **.MOD** nn [nn = 2, 4 or 8]

Comment: #<Any text string>
Label: Any alphanumeric string, starting with an alpha character

Restrictions:

Each comparator is limited as to which constant data registers may be used, according to Table 0-1 below:

Comparator #	Mask	Data
0	0,1,2,3,4,5,6	0,1,2,3,4,5,6
1	0,1,2,3,4,5,6	3,4,5,6,7,8,9
2	0,1,2,3,4,5,6	6,7,8,9,10,11,12
3	0,1,2,3,4,5,6	8,9,10,11,12,13,14

Table 0-1: Comparator Constant Restrictions

Up to 2 **Holding_Register** assignments may be made per Microcode word. Only one of them may use the "*4" operation.

Each arithmetic unit feeding the holding registers is limited as to which constant data registers may be used, according to Table 0-2 below (any of the 4 holding registers may also be selected as data input):

AU #	Mask	Data
0	0,1,2,3,4,5,6	0,1,2,3,4,5,6,7,8,9,10,11
1	0,1,2,3,4,5,6	3,4,5,6,7,8,9,10,11,12,13,14

Table 0-2: Arithmetic Unit Constant Restrictions

The hard compares are grouped into 8 groups (MAC, VLAN, SNAP, ARP, IPV4, UDP and ICMP). Within a single Microcode instruction only compares from one of these groups may be selected.

The **Flag_Register** and the **Event_Flag** inputs are tied together, so if **Flag_Register(*n*)** is set in a given Microcode instruction, then if **Event_Flag(*n*)** is also set, it must receive the same value. These inputs are restricted as to which conditions and gate outputs may be selected, so it is necessary to

carefully choose the use of each flag bit so that it can be set properly. These restrictions are shown in Table 0-3 below. The Hard_Compares are grouped into 8 groups. Hard_Compare(n) indicates that the n'th Hard_Compare ("Index" = n) in any group in Table 0-5 may be used as that flag's input. For example, Hard_Compare(0) means that any of the following hard compares may be used: **MAC_BCAST_CMP, VLAN_IPV4_TYPE_CMP, SNAP_UI_OUI_CMP, ARP_PTCL_CMP, IPV4_VERSION_CMP, UDP_TOTLEN0_CMP, ICMP_TOTLEN0_CMP** or **CA_CSUM_RSLT**.

Flag #	Possible inputs
0	Hard_Compare(0), **Soft_Compare(0), AND(0), CA_CSUM_RSLT**
1	Hard_Compare(1), **Soft_Compare(1), AND(0), OR(0)**
2	Hard_Compare(3), **Soft_Compare(3), AND(1), AND(2)**
3	Hard_Compare(6), **Soft_Compare(2), AND(3), OR(1)**
4	Hard_Compare(0), **Soft_Compare(0), Soft_Compare(3), AND(2)**
5	Hard_Compare(1), **Hard_Compare(5), Soft_Compare(1), AND(3)**
6	Hard_Compare(9), **Soft_Compare(2), AND(1), OR(0)**
7	Hard_Compare(5), **Soft_Compare(0), AND(0), AND(2)**
8	Hard_Compare(3), **Soft_Compare(2), Soft_Compare(3), OR(1)**
9	**Soft_Compare(0), Soft_Compare(2), AND(1), AND(3)**
10	Hard_Compare(7), **Soft_Compare(1), AND(0), OR(0)**
11	Hard_Compare(8), **Soft_Compare(2), AND(1), AND(2)**
12	Hard_Compare(9), **Soft_Compare(3), AND(2), OR(1)**
13	**Soft_Compare(0), Soft_Compare(2), AND(3), OR(0)**
14	**Soft_Compare(1), Soft_Compare(3), AND(0), AND(1)**
15	Hard_Compare(10), **Soft_Compare(0), AND(3), OR(1)**

Table 0-3: Flag input restrictions

The **Abort** gate inputs are limited as to which compares, gates and flag registers may be used, according to Table 0-4 below:

Abort input #	Possible Inputs
0	Hard_Compare(0), Hard_Compare(1), **Soft_Compare(0), Soft_Compare(1), AND(0), AND(1), OR(0)**
1	Hard_Compare(4), **Soft_Compare(2), Soft_Compare(3), AND(2), AND(3), OR(1), Flag_Register(7)**
2	Hard_Compare(3), **Soft_Compare(0), Soft_Compare(3), AND(1), AND(3), OR(0), Flag_Register(8)**
3	Hard_Compare(5), **Soft_Compare(1), Soft_Compare(2), AND(2), OR(1), Flag_Register(9), Flag_Register(15)**

Table 0-4: Abort Gate Input Restrictions

The Event_Type and Event_Size assignments are mutually exclusive.

The Event_Subcode and Option_Offset assignments are mutually exclusive.

Notes:

All operations are optional (in spite of the missing "{}" brackets). If the **Next_Address** operation is not used, the assembler will assume a branch to the following instruction.

gate_list: {**NOT**} Compare_Result {gate_operator {**NOT**} Compare_Result {gate_operator {**NOT**} Compare_Result {gate_operator {**NOT**} Compare_Result}}}

gate_operator: **&&** | **||**

abort_list: {**NOT**} (Compare_Result | AND(a) | OR(b) | Flag_Register(c)) { || {**NOT**} (Compare_Result | AND(d) | OR(e) | Flag_Register(f)) { || {**NOT**} (Compare_Result | AND(g) | OR(h) | Flag_Register(i)) { || {**NOT**} (Compare_Result | AND(j) | OR(k) | Flag_Register(l))}}}
[a,d,g,j = 0-3, b,e,h,k = 0-1, c,f,I,l = 0-15]

Compare_Result: **Soft_Compare(*c*)** | Hard_Compare [*c*=0-3]

Hard_Compare: In addition to the soft comparators (**Soft_Compare**), there are additional comparators which are hard-wired to certain LHB locations, fixed mask values or Constant Mask Registers, and fixed compare values or Constant Data Registers. These are named according to their intended use, although any of them may be used at any time if useful. Table 0-5 below describes them.

In the "LHB Bytes" column, "0" refers to the first/newest/upper-most byte in the LHB, while "15" would refer to the last/oldest/lowest byte. A bit range after the byte number, such as [7:4], indicates that only those bits of the indicated byte are used in the comparison.

In the Compare Equation, a hex value indicates that the LHB is compared to a hard-wired value which cannot be changed. A bold **NAME** indicates that a specific Constant Data Register is used, so the proper value must be loaded into that Constant Data Register along with the Microcode.

Name	Group	Index	LHB Bytes	Compare Equation
MAC_ETHER_TYPE_CMP	0	0	12-13	LHB > 0x05DC
MAC_MCAST_CMP	0	1	0-5	LHB DA bit 40 = 1
TMAC_UCAST_CMP	0	2	0-5	(LHB & **TMAC_MASK**) == **EXP_TMAC_DA**
PMAC_UCAST_CMP	0	3	0-5	(LHB & **PMAC_MASK**) == **EXP_PMAC_DA**
MAC_IPV4_TYPE_CMP	0	4	12-13	LHB == 0x0800
MAC_ARP_TYPE_CMP	0	5	12-13	LHB == 0x0806
MAC_VLAN_TYPE_CMP	0	6	12-13	LHB == 0x8100
MAC_SAP_CMP	0	7	14-15	LHB == 0xAAAA
MAC_IPV6_TYPE_CMP	0	8	12-13	LHB == 0x86DD
MAC_BCAST_CMP	0	9	0-5	LHB == 0xFFFFFFFFFFFF
MAC_FRAME_ERROR	0	10	NA	Header Buffer Frame Error
VLAN_ETHER_TYPE_CMP	1	0	2-3	LHB > 0x05DC
VLAN_IPV4_TYPE_CMP	1	1	2-3	LHB == 0x0800
VLAN_ARP_TYPE_CMP	1	2	2-3	LHB == 0x0806
VLAN_SAP_CMP	1	3	4-5	LHB == 0xAAAA
VLAN_IPV6_TYPE_CMP	1	4	2-3	LHB == 0x86DD
VLAN_CFI_CMP	1	5	0[4]	LHB = 1
SNAP_UI_OUI_CMP	2	0	2-5	LHB == 0x03000000
SNAP_IPV4_TYPE_CMP	2	1	6-7	LHB == 0x0800
SNAP_ARP_TYPE_CMP	2	2	6-7	LHB == 0x0806
SNAP_IPV6_TYPE_CMP	2	3	6-7	LHB == 0x86DD
ARP_PTCL_CMP	3	0	2-3	LHB == **EXP_PROTOCOL**
ARP_L2ALEN_CMP	3	1	4	LHB == 0x06
ARP_L3ALEN_CMP	3	2	5	LHB == **EXP_L3A_LEN**
ARP_OP0_CMP	3	3	6-7	LHB == 0x0001
ARP_OP1_CMP	3	4	6-7	LHB == 0x0002
IPV4_VERSION_CMP	4	0	0[7:4]	LHB == 0x4
IPV4_HDRLEN_CMP	4	1	0[3:0]	LHB < 0x5
IPV4_OPTION_CMP	4	2	0[3:0]	LHB > 0x5
IPV4_TOTLEN_CMP	4	3	2-3	LHB < (HDR_LEN * 4)
IPV4_UDP_TYPE_CMP	4	4	9	LHB == 0x11
IPV4_LASTFRAG_CMP	4	5	6-7	(B6[6] == 0) && {B6[4:0], B7} != 0
IPV4_FRAG_CMP	4	6	6-7	(B6[6] == 1) || {B6[4:0], B7} != 0
IPV4_ICMP_TYPE_CMP	4	7	9	LHB == 0x01
IPV4_TCP_TYPE_CMP	4	8	9	LHB == 0x06
IPV4_FIRSTFRAG_CMP	4	9	6-7	(B6[6] == 1) && {B6[4:0], B7} == 0
IPV4_FRAGOFFSET_CMP	4	10	6-7	{B6[4:0], B7} == 0
UDP_TOTLEN2_CMP	5	0	NA	HREG(0) > 0x0014
UDP_HDRLEN_CMP	5	1	4-5	LHB < 0x0008
UDP_TOTLEN1_CMP	5	2	NA	HREG(0) < 0x0005
UDP_TOTLEN0_CMP	5	3	4-5	LHB > HREG[0]
UDP_CSUM_CMP	5	6	6-7	LHB == 0
ICMP_TOTLEN0_CMP	6	0	NA	HREG(0) < 0x000D
ICMP_HDRLEN_CMP	6	1	NA	HREG(0) < 0x0004
ICMP_TOTLEN1_CMP	6	2	NA	HREG(0) > 0x001D
CA_RSLT_VALID	7	0	NA	Checksum result
UNKNOWN_OPTION	7	3	NA	Unknown Option Bit

Table 0-5: Hard Comparators

Semantics:

Done
This indicates that the microcode is requesting the next packet header from the Header Buffer. This should be asserted prior to the last microcode instruction such that it ensures that information for the next packet is present on the Header Buffer Interface the same clock in which the packet processor can use it. A delayed version of this signal is used to clear various registers (such as the Event, Pseudo-header, etc), and assert "ip_frm_end" on the Event Queue Interface.

Drop_Frame
This frame is to be dropped. Drives Header Buffer Interface pin "drop_frame".

Soft_Compare(*i*) := (LHB_Start_Byte(*j*) | Holding_Register(*k*) {&& Constant_Mask(*l*)}
(== | < | >) Constant_Data(*m*)
This specifies the inputs to and configuration of General Purpose 16-bit Comparator "i". The inputs are data, mask and constant. The data input is selected from either the LHB (specifying LHB Start Byte "j" will actually select Byte "j" and Byte "j+1") or Holding Register "k". An optional mask "l" can be applied to the selected data. The selected, masked data is then compared to constant "m" as specified by the operator (==, <, >) to produce a single bit compare result.

AU(*ii*) := (LHB_Start_Byte(*jj*) | Holding_Register(*kk*) | <16-bit field>) { && **Constant_Mask(*ll*)}**
{ *4 | (+ | -) (**Constant_Data(*mm*) | Holding_Register(*nn*)**)
This specifies the inputs to and configuration of 16-bit Arithmetic Unit "ii". The inputs are data, mask and constant. The data input is selected from either the LHB (specifying LHB Start Byte "jj" will actually select Byte "j" and Byte "j+1") or Holding Register "kk" or the IMMEDIATE_DATA field in the microcode. An optional mask "ll" can be applied to the selected data. The selected, masked data is then compared to constant "mm" or Holding Register "nn" as specified by the operator (*4, +, -) to produce a 16-bit arithmetic result.

Holding_Register(*pp*) := AU(*qq*)
This specifies that the 16-bit output of Arithmetic Unit "*qq*" should be stored into Holding Register "*pp*".

AND(*m*) := gate_list,
Up to 4 compare results (specified by gate_list) are ANDed to produce a single bit result.

OR(*n*) := gate_list,
Up to 4 compare results (specified by gate_list) are ORed to produce a single bit result.

ER_Flag_Register := *rr,*
The flag portion of event register "*rr*" (eventually destined for the Event structure) is to be loaded. This indicates which word to load. The bit values are defined by the **Event_Flag** command.

Flag_Register(*p*) := (**AND(*q*)** | Compare_Result | OR(r) | CA_RSLT_VALID),
Flag register "*p*" (one of the 16 flag flip-flops) is to be loaded with the result of **AND** gate "*q*", the result of **OR** gate "r", the result of the checksum accumulator or the result of one of the comparators, which can be either a hard-wired comparator or one of the 4 soft comparators. Note that, for a given index "*p*", if a **Flag_Register** and an **Event_Flag** are both loaded, they will be loaded with the same value.

Event_Flag(*p*) := (**AND(*q*)** | Compare_Result | OR(r) | CA_RSLT_VALID),
Flag big "*p*" in the event register specified by the **ER_Flag_Register** command is to be loaded as described in the **Flag_Register** command.

Load_Create_Flow_Flag
Writes the "Create_Flow_Flag" into bit 127 of event register #1.

Host_Packet
Indicates that the current packet type is "host". This control the data source selection for the "Create Flow" flag, Event Type field, Flow Key and Event Size.

Event_Type := <6-bit field>
Causes the 6-bit value specified to be written into bits 121:116 of event register #0.

Load_Flow_Key_ReceiveIF_Upper
This writes bits 11:8 of the Receive Interface into bits 99:96 of event register #0.

Load_Event_Frame_ID
This writes bits 7:0 of the Receive Interface into bits 127:120 of event register #1.

Load_Event_IPU_SPI4_Port_Nums
This writes IPU Number (input pin IPU_ID) and SPI4 Port Number from the header side info into bits 87:80 of event register #2.

Load_Event_L3_Header_Offset
This writes L3 Header Offset bits (taken from the Bytes Popped Register) into bits 71:64 of event register #2.

Load_Event_Size
This writes the Event Size into bits 124:120 of event register #1.

Event_Subcode := <8-bit field>
This writes the specified 8-bit value into bits 119:112 of event register #1.

Load_Event_Payload_Scratch_Offset
This writes the scratch payload offset into bits 95:88 of event register #1.

Load_Flow_Key_Protocol
This writes the Layer 3 Protocol field into bits 119:112 of event register #1.

Load_Flow_Key_SP_DP
This writes the SP/DP field into bits 95:64 of event register #1.

Load_Flow_Key_SA
This writes the SA field into bits 63:32 of event register #1.

Load_Flow_Key_DA
This writes the DA field into bits 31:0 of event register #1.

VLAN_Tag := (LHB | PVID)
This writes either LHB bits 127:112 or the PVID into bits 111:96 of event register #2.

Load_Switch_Fabric_Header_Length
This writes the Switch Fabric Header Length into bits 31:24 of event register #1.

Inc_Counter(*cc***)**
Increments count #*cc*. There are 7 counters, numbered 1-7.

Abort := gate_list
There are several soft conditions which can be created. Abort is a special condition, in that it is hard-wired to be the highest-priority input to the branch condition priority encoder, and in that it can be the

logical OR of up to 4 other conditions, each of which may be a hard condition, a soft condition, or the output of one of the 4 condition AND gates or the 2 condition OR gates.

LHB_Pop *constant*
This pops from 0-16 bytes off the LHB. The HB (Header Buffer) pop control logic monitors this, and will load 16 bytes from the HB if the **LHB_Pop** will create room for 16 bytes. This load occurs simultaneously with the pop, so it can be guaranteed that at least 16 bytes of valid packet data is always present in the LHB.
This has the side-effect of adding to the checksum whatever bytes were popped off of the LHB.

Pseudo_Header_Word(*u*) := {<4-bit field> **&&**} (**LHB_Word(*v*) | Holding_Register(*w*)**)
This can load any word of the pseudo-header register with any of the 4 words in the LHB or any of the 4 holding registers. A 4-bit mask can be applied to give byte-level write control.

CheckSum_Accumulate := **LHB | Pseudo_Header**
Normally bytes popped off the LHB are added to the checksum. However, this command can be used to add the contents of the pseudo-header register to the checksum.

CheckSum_Clear
This clears the running checksum, so a new checksum can be accumulated.

IP_Option_Length = **AU(*x*)**
This causes the output of one of the Arithmetic Units to be written into bits 106:103 of event register #3.

TCP_Option_Length = AU(*y*)
This causes the output of one of the Arithmetic Units to be written into bits 106:103 of event register #6.

Option_Parse (IP | TCP)
This command fires off the Option Parsing Unit, which is an independent state-machine which parses the IP or TCP options. Which type of options to parse is indicated by the argument (**IP** or **TCP**).

Event_Queue(*aa*) := { <4-bit word-mask> **&&** } (**LHB_Word_Shift(*bb*) | Event_Register(*cc*)**)
Writes **Event_Queue** *#aa* with a quad-word from either the LHB or and event register. Data from the LHB can be masked at the word level to prevent writing of that word.

Next_Address := Label
Causes a jump (branch) to the instruction indicated by the label. If a conditional branch is requested via the **Branch_Condition** command, then "Label" is the starting address of a table of instructions which are the target of the conditional branch.
Conditional branches are implemented by replacing the least-significant bits of the address with a 3-bit code. This code can be the result of prioritization of up to 6 branch conditions, or it can be up to 3 branch conditions (**Branch_Case**).
Thus, it is necessary for "Label" to be at a MOD nn boundary, to make certain none of the least-significant bits are true.
The following applies if **Branch_Case** is not enabled:
If no conditions are true, the instruction at "Label" will be executed next.
If the Abort condition is true, the instruction at "Label"+1 will be executed next.
If the most significant condition below Abort is true (**Branch_Condition(0)**), the instruction at "Label"+2 will be executed next.
If the least significant condition is true (**Branch_Condition(5)**), the instruction at "Label"+7 will be executed next.

Branch_Condition(*qq*) := (**AND(*rr*) | OR(*ss*) | Flag_Reg(*tt*)**
| Compare_Result | **Start_Address_Offset**)

This specifies the source of a branch condition, of which there are 6 (0-5). Along with theAbort condition, they are usually fed into a priority encoder to produce a 3-bit result, which becomes the 3 least-significant bits of the next microcode address.

Branch_Case
This disables the priority encoder, causing a "case" type of branch (case statement).
Branch_Conditions 0, 1 & 2 become the 3 least-significant bits of the next microcode address.